

# Dynamic Programming

---

MATRIX CHAIN MULTIPLICATION

# Matrix Chain Multiplication Problem

Multiplying non-square matrices:

- $A$  is  $p \times q$ ,  $B$  is  $q \times r$
- $AB$  is  $p \times r$  whose  $(i, j)$  entry is  $\sum a_{ik} b_{kj}$

Must be equal

Computing  $AB$  takes  $p \cdot q \cdot r$  scalar multiplications and  $p(q-1)r$  scalar additions (using basic algorithm).

Suppose we have a sequence of matrices to multiply. What is the best order?

# Matrix Chain Multiplication Problem

Given a sequence of matrices  $A_1, A_2, \dots, A_n$ , then

Compute  $C = A_1 \cdot A_2 \cdot \dots \cdot A_n$

Different ways to compute  $C$

$$C = (A_1 A_2)((A_3 A_4)(A_5 A_6))$$

$$C = (A_1 (A_2 A_3)(A_4 A_5)) A_6$$

- Matrix multiplication is associative
  - So output will be the same
- However, time cost can be very different
  - Example

# Why Order Matters

---

Suppose we have 4 matrices:

- $A, 30 \times 01$
- $B, 01 \times 40$
- $C, 40 \times 10$
- $D, 10 \times 25$

$((AB)(CD))$  : requires 41,200 multiplications

$$[ (30 \times 1 \times 40) + (40 \times 10 \times 25) + (30 \times 40 \times 25) = 41,200 ]$$

$(A((BC)D))$  : requires 1400 multiplications

$$[ (1 \times 40 \times 10) + (1 \times 10 \times 25) + (30 \times 1 \times 25) = 1,400 ]$$



# Matrix Chain Multiplication Problem

Given a sequence of matrices  $A_1, A_2, \dots, A_n$ , where  $A_i$  is  $p_{i-1} \times p_i$ :

- What is minimum number of scalar multiplications required to compute  $A_1 \cdot A_2 \cdot \dots \cdot A_n$ ?
- What order of matrix multiplications achieves this minimum?
- Fully parenthesize the product in a way that minimizes the number of scalar multiplications

(( )( ))(( )( ( )( )( )))

No. of parenthesizations: ???

# A Possible Solution

---

Try all possibilities and choose the best one.

Drawback is there are too many of them (exponential in the number of matrices to be multiplied)

The number of parenthesizations is

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

The solution to the recurrence is  $\Omega(2^n)$

No. of parenthesizations: **Exponential**

Need to be more clever - try dynamic programming !

# Step 1: Optimal Substructure Property

---

A problem exhibits *optimal substructure* if an optimal solution to the problem contains within it optimal solutions to subproblems.

Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.

Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

We must also take care to ensure that the total number of distinct subproblems is a polynomial in the input size.

# Step 1: Optimal Substructure Property

---

Define  $A_{i..j}$ ,  $i \leq j$ , to be the matrix that results from evaluating the product  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ .

If the problem is nontrivial, i.e.,  $i < j$ , then to parenthesize  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ , split the product between  $A_k$  and  $A_{k+1..j}$  for some  $k$ , where  $i \leq k < j$ .

- The cost of parenthesizing this way is
  - The cost of computing the matrix  $A_{i..k}$  +
  - The cost of computing the matrix  $A_{k+1..j}$  +
  - The cost of multiplying them together

- The optimal substructure of this problem is:

An optimal parenthesization of  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$  contains within it optimal parenthesizations of  $A_i \cdot A_{i+1} \cdot \dots \cdot A_k$  and  $A_{k+1} \cdot A_{k+2} \cdot \dots \cdot A_j$

**Proof ?**

# Overlapping Subproblem Property

---

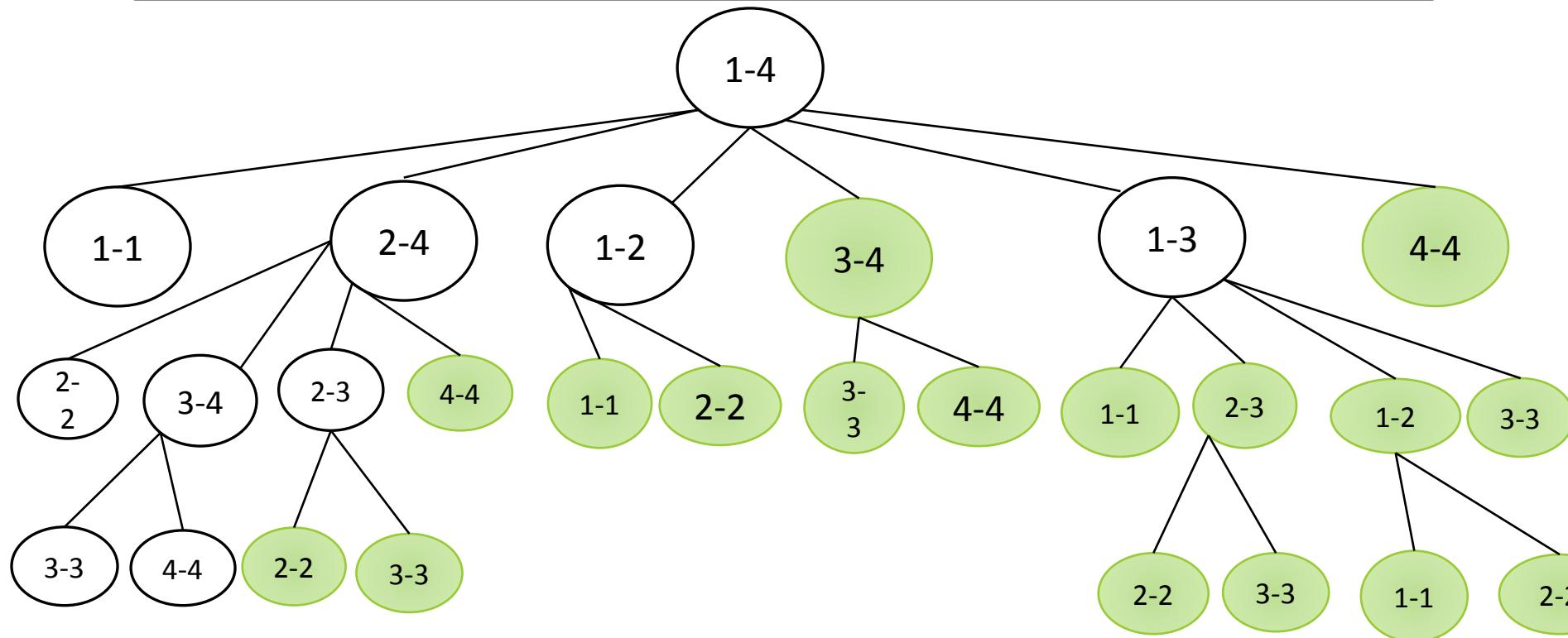
Two subproblems of the same problem are **independent** if they do not share resources.

Two subproblems are ***overlapping*** if they are really the same subproblem that occurs as a subproblem of different problems.

A problem exhibits ***overlapping subproblem*** if the number of subproblems is “small” in the sense that a recursive algorithm solves the same subproblems over and over, rather than always generating new subproblems.

Dynamic programming algorithms take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed.

# Overlapping Subproblem Property



# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

5x4 4x6 6x2 2x7

m represents cost of multiplication

$$A_1 \times A_1 \quad m[1,1] = 0;$$

$$A_2 \times A_2 \quad m[2,2] = 0;$$

$$A_3 \times A_3 \quad m[3,3] = 0;$$

$$A_4 \times A_4 \quad m[4,4] = 0;$$

m	1	2	3	4
1	0			
2		0		
3			0	
4				0

s	1	2	3	4
1				
2				
3				
4				

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

$$5 \times 4 \quad 4 \times 6 \quad 6 \times 2 \quad 2 \times 7$$

**m** represents cost of multiplication

$$A_1 \times A_2$$

$$m[1,2] = 5 \times 6 \times 4 = 120;$$

$$A_2 \times A_3$$

$$m[2,3] = 4 \times 2 \times 6 = 48;$$

$$A_3 \times A_4$$

$$m[3,4] = 6 \times 7 \times 2 = 84;$$

m	1	2	3	4
1	0	120		
2		0	48	
3			0	84
4				0

s	1	2	3	4
1		1		
2			2	
3				3
4				

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

5x4 4x6 6x2 2x7

**m** represents cost of multiplication

$$A_1 \times A_2 \times A_3$$

$$A_1 (A_2 \cdot A_3)$$

$$\begin{aligned} & m[1,1] + m[2,3] + d_0 d_1 d_3 \\ & = 0 + 48 + 5 \times 4 \times 2 = 88; \end{aligned}$$

$$(A_1 \cdot A_2) \cdot A_3$$

$$\begin{aligned} & m[1,2] + m[3,3] + d_0 d_2 d_3 \\ & = 120 + 0 + 5 \times 6 \times 2 = 180; \end{aligned}$$

m	1	2	3	4
1	0	120	88	
2		0	48	
3			0	84
4				0

s	1	2	3	4
1		1	1	
2			2	
3				3
4				

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

5x4 4x6 6x2 2x7

**m** represents cost of multiplication

$$A_2 \times A_3 \times A_4$$

$$A_2 (A_3 \cdot A_4)$$

$$m[2,2] + m[3,4] + d_1 d_2 d_4$$

$$= 0 + 84 + 4 \times 6 \times 7 = 252;$$

$$(A_2 \cdot A_3) \cdot A_4$$

$$m[2,3] + m[4,4] + d_1 d_3 d_4$$

$$= 48 + 0 + 4 \times 2 \times 7 = 104;$$

m	1	2	3	4
1	0	120	88	
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	
2			2	3
3				3
4				

# Matrix Chain Multiplication

Suppose You have 04 Matrices

$$A_1 \quad A_2 \quad A_3 \quad A_4$$

$$5 \times 4 \quad 4 \times 6 \quad 6 \times 2 \quad 2 \times 7$$

**m** represents cost of multiplication

$$A_1 \times A_2 \times A_3 \times A_4$$

$$A_1 (A_2 A_3 \cdot A_4)$$

$$m[1,1] + m[2,4] + d_0 d_1 d_4$$

$$= 0 + 104 + 5 \times 4 \times 7 = 244;$$

$$(A_1 \cdot A_2) (A_3 \cdot A_4)$$

$$m[1,2] + m[3,4] + d_0 d_2 d_4$$

$$= 120 + 84 + 5 \times 6 \times 7 = 414;$$

$$(A_1 \cdot A_2 \cdot A_3) \cdot A_4$$

$$m[1,3] + m[4,4] + d_0 d_3 d_4$$

$$= 88 + 0 + 5 \times 2 \times 7 = 158;$$

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

# Now The Formula

---

- Define  $m[i, j]$  to be the minimum number of multiplications needed to compute  $A_i \cdot A_{i+1} \cdot \dots \cdot A_j$ .
  - Goal: Find  $m[1, n]$
  - Basis:  $m[i, i] = 0$
  - Recursion: How to define  $m[i, j]$  recursively ?
- Consider all possible ways to split  $A_i$  through  $A_j$  into two pieces.
- Compare the costs of all these splits:
  - best case cost for computing the product of the two pieces
  - plus the cost of multiplying the two products
- Take the best one

# Now The Formula

---

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & i < j \end{cases}$$

# Step 2: Develop a Recursive Solution

Matrix\_Chain\_RecursiveWay (m, i, j)

1. for (i <- 0 to n-1)
2.     for (j <- i to n-1)  
           m[ i, j ] <-  $\infty$
3.     if(i=j)  
4.         then return 0;
5.     if(m[ i, j ]  $\neq \infty$ )  
6.         then return m[i, j];
7.     for (k<- i to j)  
8.         do cost= Matrix\_Chain\_RecursiveWay (m, i, k) +  
                  Matrix\_Chain\_RecursiveWay (m, k+1, j) + d<sub>i-1</sub>d<sub>k</sub>d<sub>j</sub>;
9.         if(cost < m[ i, j])  
10.             then m[ i, j ] <- cost;
11.     return m [ i, j];



Memorization

running time  
 $O(n^3)$

## Step 2: Develop a Recursive Solution

---

- Let  $T(n)$  be the time taken by Recursive-Matrix-Chain for  $n$  matrices.  $T(1) \geq 1$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1$$

For  $i = 1, 2, \dots, n-1$ , each term  $T(i)$  appears once as  $T(k)$  and once as  $T(n-k)$ . Thus, we have

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} T(i) + n \\ &\geq 2^{n-1} \end{aligned}$$

Then  $T(n) = \Omega(2^n)$

# Step 3: Compute the Optimal Costs

Find Dependencies among Subproblems

---

$m:$	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

← GOAL

computing the red square requires the blue ones: to the left and below.

# Step 3: Compute the Optimal Costs

## Find Dependencies among Subproblems

$m:$	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

- Computing  $m(i, j)$  uses
  - everything in same row to the left:  
 $m(i, i), m(i, i+1), \dots, m(i, j-1)$
  - and everything in same column below:  
 $m(i+1, j), m(i+2, j), \dots, m(j, j)$

# Step 3: Compute the Optimal Costs

## Identify Order for Solving Subproblems

- Solve the subproblems (i.e., fill in the table entries) this way:
  - go along the diagonal
  - start just above the main diagonal
  - end in the upper right corner (goal)

*m:*

	1	2	3	4	5
1					
2	n/a				
3	n/a	n/a			
4	n/a	n/a	n/a		
5	n/a	n/a	n/a	n/a	0

GOAL

# Step 4: Construct an Optimal Solution

---

- It's fine to know the cost of the cheapest order, but what is that cheapest order?
- Keep another array  $s$  and update it when computing the minimum cost in the inner loop
- After  $m$  and  $s$  have been filled in, then call a recursive algorithm on  $s$  to print out the actual order

`Print_order(s, i, j)`

```
1. if (i=j)
   then print "A"i;
Else
  print "("
  Print_order (s, i, s[i,j] );
  Print_order (s, s[i,j] +1, j);
  print ")";
```

---

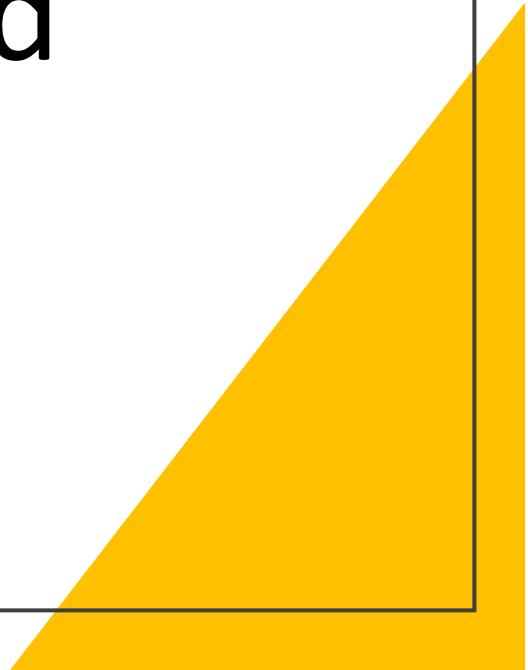
Thank You!

# CSE 215

# Data Structures and

# Algorithms-II

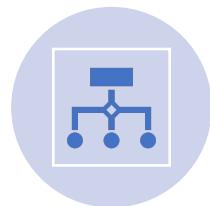
Branch and Bound



# Branch N Bound



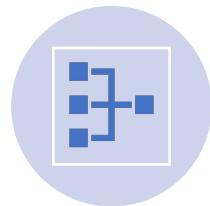
Branch and bound is an algorithm design paradigm for discrete and combinatoric optimisation problems, as well as mathematical optimisation.



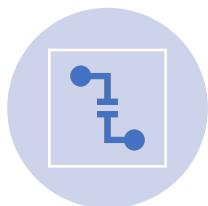
A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions.



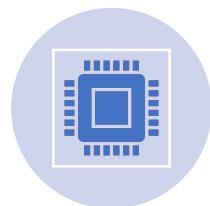
That is, the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.



The algorithm explores branches of this tree, which represent the subsets of the solution set.



Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution.



And is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

# Backtracking vs Branch N Bound



Parameter	Backtracking	Branch and Bound
Approach	Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.	Branch-and-Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution than the pre-solution leads to, it abandons that pre-solution. It completely searches the state space tree to get optimal solution.
Traversal	Backtracking traverses the state space tree by <a href="#">DFS(Depth First Search)</a> manner.	Branch-and-Bound traverse the tree in any manner, <a href="#">DFS</a> or <a href="#">BFS</a> .
Function	Backtracking involves feasibility function.	Branch-and-Bound involves a bounding function.
Problems	Backtracking is used for solving Decision Problem.	Branch-and-Bound is used for solving Optimisation Problem.
Searching	In backtracking, the state space tree is searched until the solution is obtained.	In Branch-and-Bound as the optimum solution may be present anywhere in the state space tree, so the tree need to be searched completely.
Efficiency	Backtracking is more efficient.	Branch-and-Bound is less efficient.
Applications	Useful in solving <a href="#">N-Queen Problem</a> , <a href="#">Sum of subset</a> , <a href="#">Hamilton cycle problem</a> , <a href="#">graph coloring problem</a>	Useful in solving <a href="#">Knapsack Problem</a> , <a href="#">Travelling Salesman Problem</a> .
Solve	Backtracking can solve almost any problem. (chess, sudoku, etc ).	Branch-and-Bound can not solve almost any problem.
Used for	Typically backtracking is used to solve decision problems.	Branch and bound is used to solve optimization problems.
Nodes	Nodes in state space tree are explored in depth first tree.	Nodes in tree may be explored in depth-first or breadth-first order.
Next move	Next move from current state can lead to bad choice.	Next move is always towards better solution.
Solution	On successful search of solution in state space tree, search stops.	Entire state space tree is search in order to find optimal solution.

# Branch and Bound (B & B)

---

- An enhancement of backtracking
  - **Similarity** : A state space tree is used to solve a problem.
  - **Difference** : Used only for optimization problems.
- 2 mechanisms:
  - A mechanism to generate branches when searching the solution space
  - A mechanism to generate a bound so that many branches can be terminated

# Branch and Bound

---

- Search the tree using a breadth-first search (FIFO branch and bound).
- Search the tree as in a BFS, but replace the FIFO queue with a stack (LIFO branch and bound).
- Replace the FIFO queue with a priority queue (least-cost (or max priority) branch and bound).

The priority of a node  $p$  in the queue is based on an estimate of the likelihood that the answer node is in the subtree whose root is  $p$ .

# Branch and Bound

---

- FIFO branch and bound finds solution closest to the root.
- Backtracking may never find a solution because tree depth is infinite (unless repeating configurations are eliminated).
- Least-cost branch and bound directs the search to parts of the space most likely to contain the answer. So it could perform better than backtracking.

# Branch and Bound

---

- The idea:

Set up a **bounding function**, which is used to compute a **bound** (for the value of the objective function) **at a node** on a state-space tree and determine **if it is promising**

- **Promising** (if the bound is better than the value of the best solution so far): expand beyond the node.
- **Nonpromising** (if the bound is no better than the value of the best solution so far): do not expand beyond the node (pruning the state-space tree).
- The search proceeds until all nodes have been solved or pruned.

# Bounding

---

- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
  - Greater than some number (lower bound)
  - Or, less than some number (upper bound)
- If we are looking for a maximal optimal (knapsack), then we need an upper bound
  - For example, if the best solution we have found so far has a profit of 12 and the upper bound on a node is 10 then there is no point in expanding the node
    - The node cannot lead to anything better than a 10

# Bounding

---

- We prune (via bounding) when:  
$$(\text{currentBestSolutionCost} \geq \text{nodeBound})$$
- This tells us that we get more pruning if:
  - The currentBestSolution is high
  - And the nodeBound is low
- So we want to find a high solution quickly and we want the highest possible upper bound
  - One has to factor in the extra computation cost of computing higher upper bounds vs. the expected pruning savings

# Branch and Bound



- It is efficient in the average case because many branches can be terminated very early.
- Although it is usually very efficient, a very large tree may be generated in the worst case.
- Many NP-hard problem can be solved by B&B efficiently in the average case; however, the worst case time complexity is still exponential.

# 0-1 Knapsack

---

- Capacity W is 10
- Upper bound is \$100  
(use fractional value)

Item	Weight	Value	Value/weight
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

# Computing Upper Bound

---

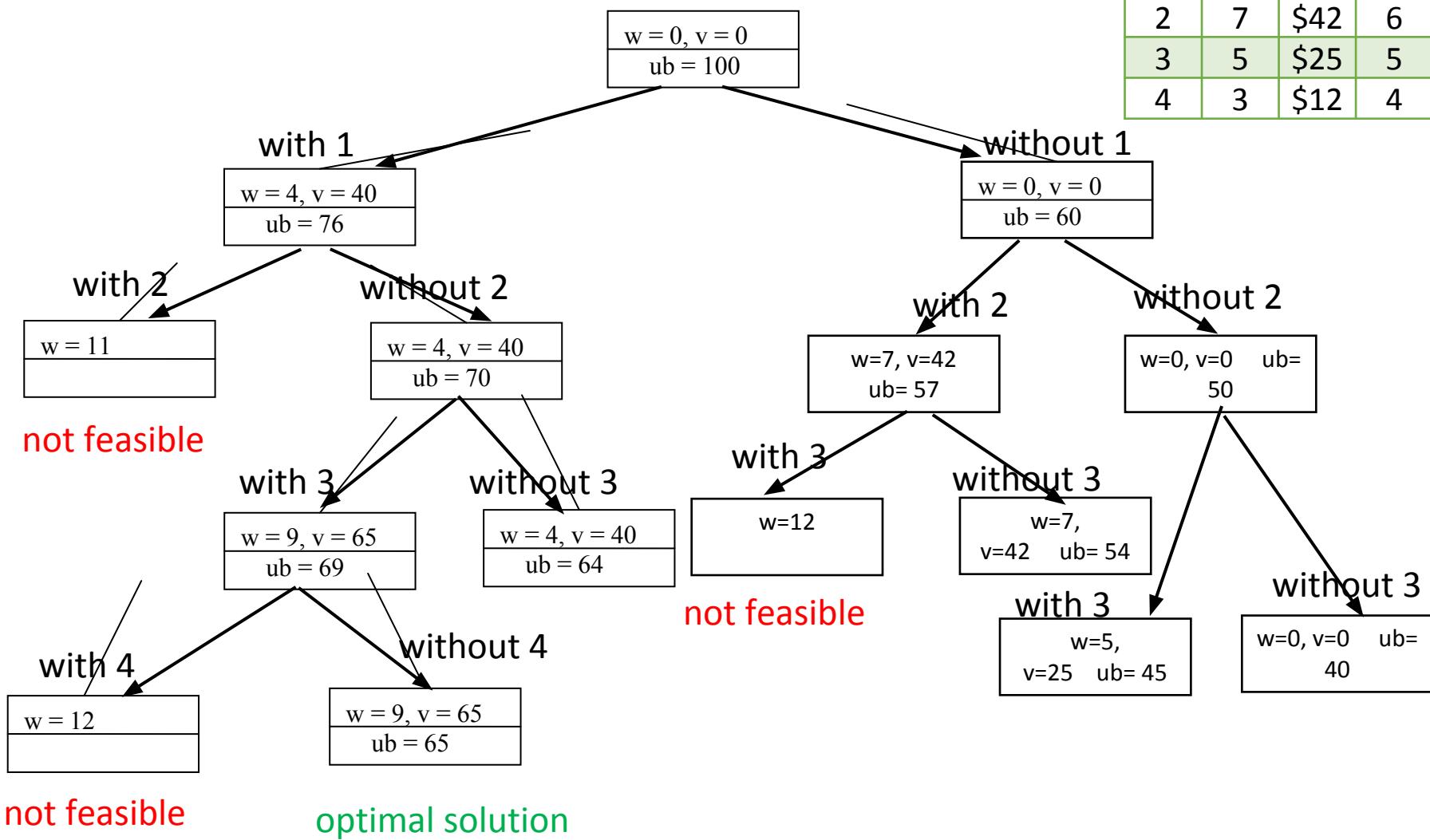
Item	Weight	Value	$V_i / W_i$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

- To compute the upper bound, we use
  - $ub = v + (W - w)(v_{i+1}/w_{i+1})$
- So the maximum upper bound is
  - pick no items, take maximum profit item
  - $ub = (10 - 0) * (\$10) = \$100$
- After we pick item 1, we calculate the upper bound as
  - all of item 1 (4, \$40) + partial of item 2 (6, \$36)
  - $\$40 + (10-4) * (\$6) = \$76$
- If we don't pick item 1:
  - $ub = (10 - 0) * (\$6) = \$60$

# State Space Tree

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$

Item	Weig ht	Valu e	Vi / Wi
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4



# Travelling Salesman Problem

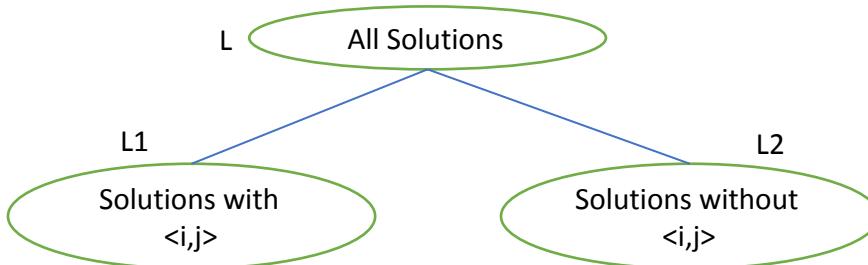
---

- Definition: Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.
- Definitions:
  - A row or column is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.
  - A matrix is reduced iff every row and column is reduced.

# Branching

- **Branching:**

- Each node splits the remaining solutions into two groups: those that include a particular edge and those that exclude that edge.
- Each node has a lower bound.
- Example: Given a graph  $G=(V,E)$ , let  $\langle i,j \rangle \in E$ ,



# Bounding

How to compute the cost of each node?

- Subtract of a constant from any row and any column does not change the optimal solution (The path).
- The cost of the path changes but the path itself.
- Let A be the cost matrix of a  $G=(V,E)$ .
- The cost of each node in the search tree is computed as follows:
  - Let R be a node in the tree and  $A^R$  its reduced matrix
  - The cost of the child  $R^S$ :
    - Set row i and column j to infinity
    - Set  $A(j,i)$  to infinity
    - Reduced S and let RCL be the reduced cost.
    - $C(S) = C^R + RCL + A(i,j)$

# Bounding cont..



**Get the reduced matrix  $A'$  of  $A$  and let  $L$  be value subtracted from  $A$**



**$L$ : represents the lower bound of the path solution**



**The cost of the path is exactly reduced by  $L$ .**



**What to determine the branching edge?**

The rule favors a solution through left subtree rather than right subtree, i.e. the matrix is reduced by a dimension.

Note that the right subtree only sets the branching edge to infinity.

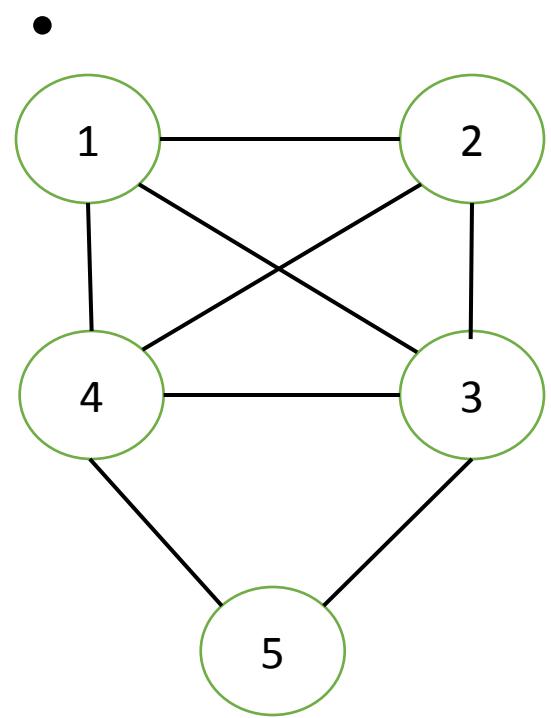
Pick the edge that causes the greatest increase in the lower bound of the right subtree, i.e. the lower bound of the root of the right subtree is greater.

# Cost Determine

Example:

- The reduced cost matrix is done as follows:
  - Change all entries of row  $i$  and column  $j$  to infinity
  - Set  $A(j,1)$  to infinity (assuming the start node is 1)
  - Reduce all rows first and then column of the resulting matrix

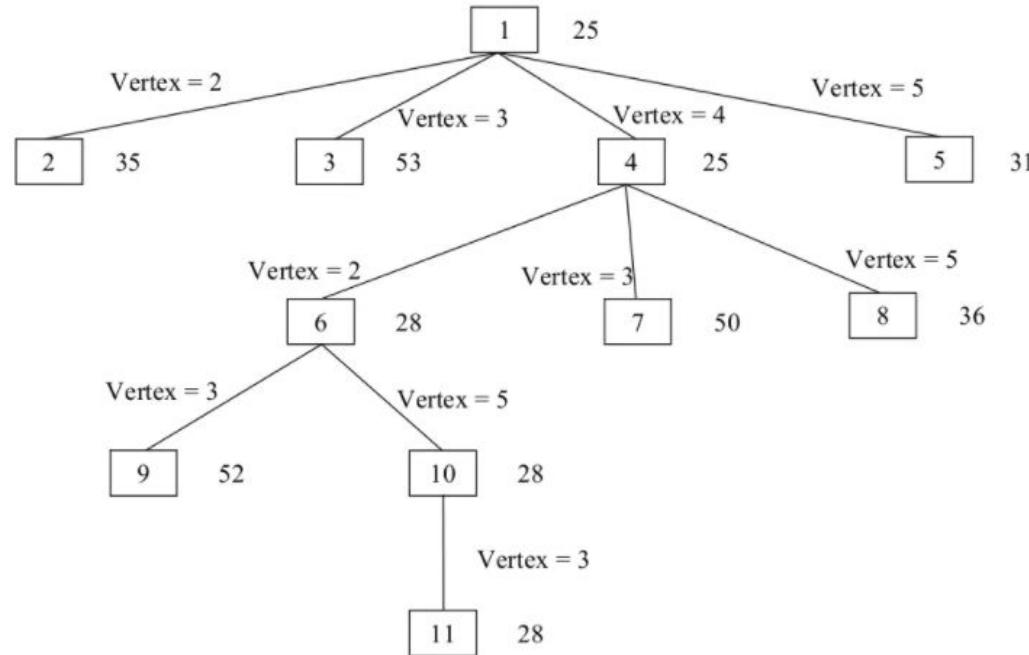
# TSP



Cost matrix

inf	20	30	10	11
15	inf	16	4	2
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

# State Space Tree



- The TSP starts from node 1: **Node 1**

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #1: reduce by 10

Before

inf	20	30	10	11
15	inf	16	4	2
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

inf	10	20	0	1
15	inf	16	4	2
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #2: reduce by 2

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 15 & \text{inf} & 16 & 4 & 2 \\ 3 & 5 & \text{inf} & 2 & 4 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 3 & 5 & \text{inf} & 2 & 4 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #3: reduce by 2

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 3 & 5 & \text{inf} & 2 & 4 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 1 & 3 & \text{inf} & 0 & 2 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #4: reduce by 3

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
19	6	18	inf	3
16	4	7	16	inf

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
16	4	7	16	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #5: reduce by 4

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
16	4	7	16	inf

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
12	0	3	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#1: reduce by 1

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
12	0	3	12	inf

inf	10	20	0	1
12	inf	14	2	0
0	3	inf	0	2
15	3	15	inf	0
11	0	3	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#2: It is reduced (**no change**)

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
12	0	3	12	inf

inf	10	20	0	1
12	inf	14	2	0
0	3	inf	0	2
15	3	15	inf	0
11	0	3	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#3: Reduce by 3

Before

inf	10	20	0	1
12	inf	14	2	0
0	3	inf	0	2
15	3	15	inf	0
11	0	3	12	inf

inf	10	17	0	1
12	inf	11	2	0
0	3	inf	0	2
15	3	12	inf	0
11	0	0	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#4: It is reduced (no change)

Before

inf	10	20	0	1
12	inf	14	2	0
0	3	inf	0	2
15	3	15	inf	0
11	0	3	12	inf

inf	10	17	0	1
12	inf	11	2	0
0	3	inf	0	2
15	3	12	inf	0
11	0	0	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#5: It is reduced (no change)

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 12 & \text{inf} & 14 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 15 & \text{inf} & 0 \\ 11 & 0 & 3 & 12 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

After

The reduced matrix is:

Cost (1) = 25

$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 2: **Node 2**
  - Cost of edge  $\langle 1,2 \rangle$  is:  $A(1,2) = 10$
  - Set row#1 = **inf** since we are choosing edge  $\langle 1,2 \rangle$
  - Set column #2= **inf** since we are choosing edge  $\langle 1,2 \rangle$
  - Set  $A(2,1) = \text{inf}$

**The resulting cost matrix is:**

The matrix is reduced:

RCL=0 (As min is 0 for all row

And columns)

The cost of node 2 (Considering Vertex 2 from vertex 1) is:

Cost(2)

$$= \text{cost}(1) + \text{RCL} + A(1,2) = 25 + 0 + 10$$

$$= 35$$

inf	inf	inf	inf	inf
inf	inf	11	2	0
0	inf	inf	0	2
15	inf	12	inf	0
11	inf	0	12	inf

- Choose to go to vertex 3: **Node 3**

- Cost of edge  $\langle 1,3 \rangle$  is:  $A(1,2) = 17$  (in the reduced matrix)
- Set row#1 = **inf** since we are staring from node 1
- Set column #3= **inf** since we are choosing edge  $\langle 1,3 \rangle$
- Set  $A(3,1) = \text{inf}$

The resulting cost matrix is:

Reduce the matrix:

- Rows are reduced
- The colum are reduced except for column # 1:
  - Reduce column 1 by 11

inf	inf	inf	inf	inf
12	inf	inf	2	0
inf	3	inf	0	2
15	3	inf	inf	0
11	0	inf	12	inf

- The lower bound is:

- RCL=11

- The cost of going through node 3 is:

- $$\begin{aligned} \text{Cost}(3) &= \text{cost}(1) + \text{RCL} + A(1,3) = 25+11+17 \\ &= 53 \end{aligned}$$

Before

inf	inf	inf	inf	inf
12	inf	inf	2	0
inf	3	inf	0	2
15	3	inf	inf	0
11	0	inf	12	inf

inf	inf	inf	inf	inf
1	inf	inf	2	0
inf	3	inf	0	2
4	3	inf	inf	0
0	0	inf	12	inf

After

- Choose to go to vertex 4: **Node 4**
  - Remember that the cost matrix is the one that was reduced at the starting vertex 1
    - Cost of edge  $\langle 1,4 \rangle$  is:  $A(1,4) = 0$
    - Set row#1 = **inf** since we are staring from node 1
    - Set column #4= **inf** since we are choosing edge  $\langle 1,4 \rangle$
    - Set  $A(4,1) = \text{inf}$
- The resulting cost matrix is:**

Reduce the matrix:

- Rows are reduced
- Columns are reduced

- The cost of going through node 4 is:
  - $\text{Cost}(4) = \text{cost}(1) + \text{RCL} + A(1,4)$
  - $= 25+0+0$
  - $= 25$

inf	inf	inf	inf	inf
12	inf	11	inf	0
0	3	inf	inf	2
inf	3	12	inf	0
11	0	0	inf	inf

- Choose to go to vertex 5: **Node 5**
  - Remember that the cost matrix is the one that was reduced at the starting vertex 1
    - Cost of edge  $\langle 1,5 \rangle$  is:  $A(1,5) = 1$
    - Set row#1 = **inf** since we are staring from node 1
    - Set column #5= **inf** since we are choosing edge  $\langle 1,5 \rangle$
    - Set  $A(5,1) = \text{inf}$
- The resulting cost matrix is:**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & 2 & \text{inf} \\ 0 & 3 & \text{inf} & 0 & \text{inf} \\ 15 & 3 & 12 & \text{inf} & \text{inf} \\ \text{inf} & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
    - Reduce Row#2: by 2
    - Reduce Row# 4: by 3
  - Columns are reduced
- The lower bound is:
  - $RCL = 2+3 = 5$
- The cost of going through node 5 is:
  - $\text{Cost}(5) = \text{cost}(1) + RCL + A(1,5)$   
 $= 25+5+1$   
 $= 31$

Reduce by 2

inf	inf	inf	inf	inf
10	inf	9	0	inf
0	3	inf	0	inf
15	3	12	inf	inf
inf	0	0	12	inf

Reduce by 3

inf	inf	inf	inf	inf
10	inf	9	0	inf
0	3	inf	0	inf
12	0	9	inf	inf
inf	0	0	12	inf

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 ↗ 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 ↗ 3
  - 4:  $\text{cost}(4) = 25$ , path: 1 ↗ 4
  - 5:  $\text{cost}(5) = 31$ , path: 1 ↗ 5
- Explore the node with the lowest cost: Node 4 has a cost of 25
- Vertices to be explored from node 4: 2, 3, and 5

- Now we are starting from the cost matrix at node 4 is:

$$\text{Cost}(4) = 25$$
$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & \text{inf} & 0 \\ 0 & 3 & \text{inf} & \text{inf} & 2 \\ \text{inf} & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 2: **Node 6** (path 1→4→2)
  - Cost of edge <4,2> is:  $A(4,2) = 3$
  - Set row#4 = **inf** since we are considering edge <4,2>
  - Set column # 2= **inf** since we are considering edge <4,2>
  - Set  $A(2,1) = \text{inf}$

- **Reduce the matrix:**

- Rows are reduced
- Columns are reduced
- The lower bound is  $RCL = 0$
- The cost of going through node 2 is:

$$\text{Cost}(6) = \text{cost}(4) + RCL + A(4,2)$$

$$= 25 + 0 + 3$$

$$= 28$$

inf	inf	inf	inf	inf
inf	inf	11	inf	0
0	inf	inf	inf	2
inf	inf	inf	inf	inf
11	inf	0	inf	inf

- Choose to go to vertex 3: **Node 7** (path 1→4→3)
  - Cost of edge <4,3> is:  $A(4,3) = 12$
  - Set row#4 = **inf** since we are considering edge <4,3>
  - Set column # 3= **inf** since we are considering edge <4,3>
  - Set  $A(3,1) = \text{inf}$

The resulting matrix is

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & \text{inf} & \text{inf} & 0 \\ \text{inf} & 3 & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & 0 & \text{inf} & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
  - Reduce Row#3:
  - Reduce column#1:
- The lower bound is:
  - $RCL = 2+11 = 13$
- Considering vertex 3 from vertex 4 is:
  - $\text{Cost}(7) = \text{cost}(4) + RCL + A(4,3)$   
 $= 25+13+12$   
 $= 50$

Reduce by 2

inf	inf	inf	inf	inf
12	inf	inf	inf	0
inf	1	inf	inf	0
inf	inf	inf	inf	inf
11	0	inf	inf	inf

inf	inf	inf	inf	inf
1	inf	inf	inf	0
inf	1	inf	inf	0
inf	inf	inf	inf	inf
0	0	inf	inf	inf

Reduce by 11

- Choose to go to vertex 5: **Node 8** (path 1→4→5)
  - Cost of edge <4,5> is:  $A(4,5) = 0$
  - Set row#4 = **inf** since we are considering edge <4,3>
  - Set column # 3= **inf** since we are considering edge <4,5>
  - Set  $A(5,1) = \text{inf}$

The resulting matrix is

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & \text{inf} & \text{inf} \\ 0 & 3 & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & 0 & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
    - Reduce Row#2
  - Columns are reduced
- The lower bound is:
  - $RCL = 11 = 11$
- Considering vertex 5 from vertex 4 is:
  - $\text{Cost}(8) = \text{cost}(4) + RCL + A(4,5)$   
 $= 25 + 11 + 0$   
 $= 36$

Reduce by 11

inf	inf	inf	inf	inf
1	inf	0	inf	inf
0	3	inf	inf	inf
inf	inf	inf	inf	inf
inf	0	0	inf	inf

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 ⊕ 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 ⊕ 3
  - 5:  $\text{cost}(5) = 31$ , path: 1 ⊕ 5
  - 6:  $\text{cost}(6) = 28$ , path: 1 ⊕ 4 ⊕ 2
  - 7:  $\text{cost}(7) = 50$ , path: 1 ⊕ 4 ⊕ 3
  - 8:  $\text{cost}(8) = 36$ , path: 1 ⊕ 4 ⊕ 5
- Explore the node with the lowest cost: Node 6 has a cost of 28
- Vertices to be explored from node 6: 3, and 5

- Now we are starting from the cost matrix at node 6 is:

**Cost(4) =28**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & 11 & \text{inf} & 0 \\ 0 & \text{inf} & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & \text{inf} & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 3: **Node 9** (path 1→4→2→3)

- Cost of edge <2,3> is:  $A(2,3) = 11$
- Set row#2 = **inf** since we are considering edge <2,3>
- Set column # 3= **inf** since we are considering edge <2,3>
- Set  $A(3,1) = \text{inf}$

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & \text{inf} & \text{inf} & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
  - Reduce Row#3
  - Reduce column# 1

• The lower bound is:

- $RCL = 2 + 11 = 13$

• Considering vertex 3 from vertex

2 is:

- $\text{Cost}(9) = \text{cost}(6) + RCL + A(2,3)$   
 $= 28 + 13 + 11$   
 $= 52$

By 2

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	0
inf	inf	inf	inf	inf
11	inf	inf	inf	inf

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	0
inf	inf	inf	inf	inf
0	inf	inf	inf	inf

- Choose to go to vertex 5: **Node 10** (path 1→4→2→5)
  - Cost of edge <2,5> is:  $A(2,5) = 0$
  - Set row#2 = **inf** since we are considering edge <2,3>
  - Set column # 3= **inf** since we are considering edge <2,3>
  - Set  $A(5,1) = \text{inf}$

- **Reduce the matrix:**

- Rows are reduced
- Columns are reduced

- The lower bound is  $RCL = 0$
- Considering vertex 3 from vertex 2 is:

$$\text{Cost}(10) = \text{cost}(6) + RCL + A(2,3)$$

$$= 28 + 0 + 0 \\ = 28$$

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
0	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	0	inf	inf

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 ⊕ 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 ⊕ 3
  - 5:  $\text{cost}(5) = 31$ , path: 1 ⊕ 5
  - 7:  $\text{cost}(7) = 50$ , path: 1 ⊕ 4 ⊕ 3
  - 8:  $\text{cost}(8) = 36$ , path: 1 ⊕ 4 ⊕ 5
  - 9:  $\text{cost}(9) = 52$ , path: 1 ⊕ 4 ⊕ 2 ⊕ 3
  - 10:  $\text{cost}(2) = 28$ , path: 1 ⊕ 4 ⊕ 2 ⊕ 5
- Explore the node with the lowest cost: Node 10 has a cost of 28
- Vertices to be explored from node 10: 3

- Now we are starting from the cost matrix at node 10 is:

**Cost(4) =28**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 0 & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 3: **Node 11** (path 1?4?2?5?3)

- Cost of edge  $\langle 5,3 \rangle$  is:  $A(5,3) = 0$
- Set row#5= **inf** since we are considering edge  $\langle 5,3 \rangle$
- Set column # 3= **inf** since we are considering edge  $\langle 5,3 \rangle$
- Set  $A(3,1) = \text{inf}$

- **Reduce the matrix:**

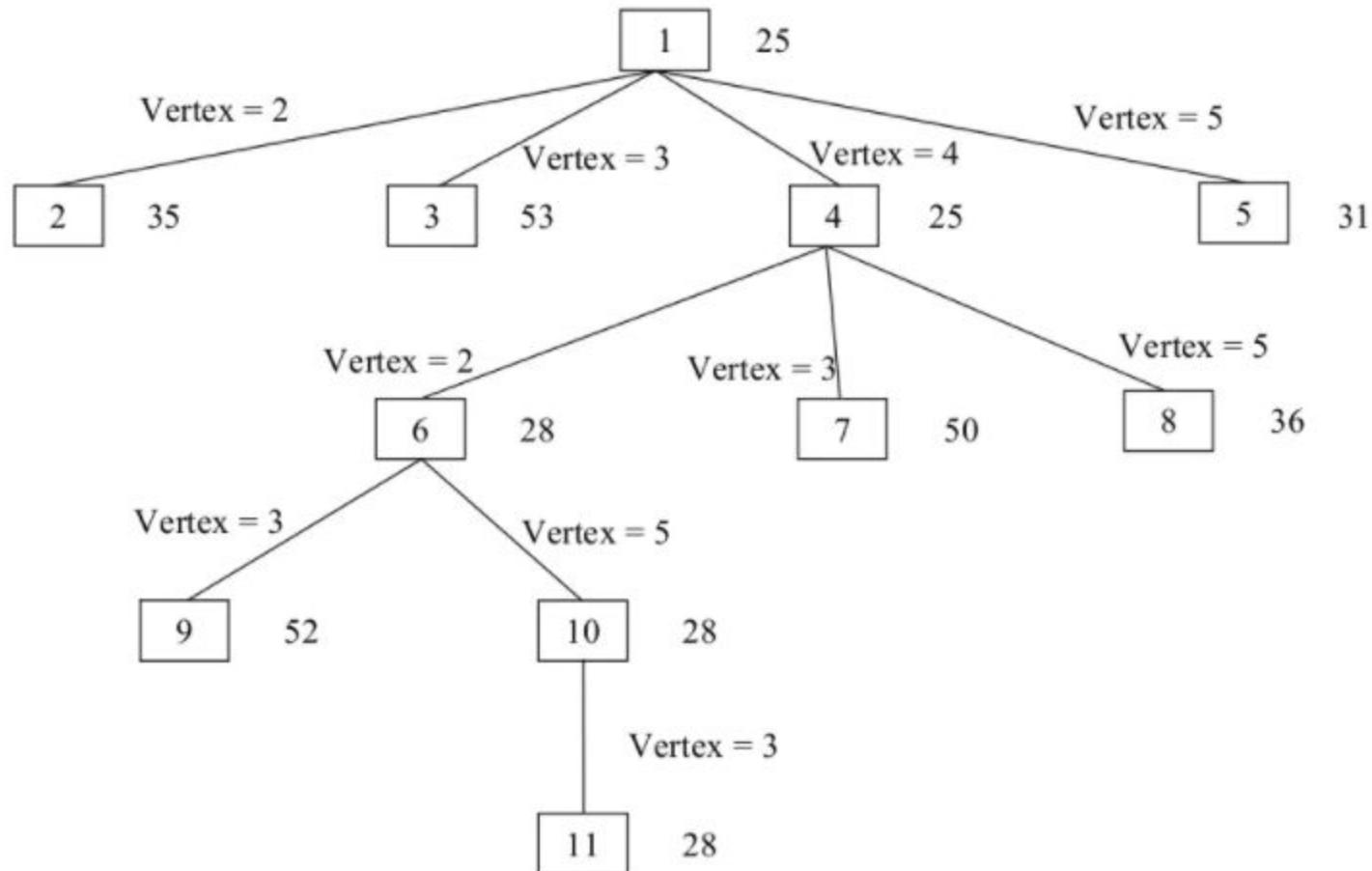
- Rows are reduced
- Columns are reduced

- The lower bound is  $RCL = 0$
- Considering vertex 5 from vertex 3 is:

$$\begin{aligned} \text{Cost}(11) &= \text{cost}(10) + RCL + A(5,3) \\ &= 28 + 0 + 0 \\ &= 28 \end{aligned}$$

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	inf

# State Space Tree



The short tour: path 1 → 4 → 2 → 5 → 3 → 1)

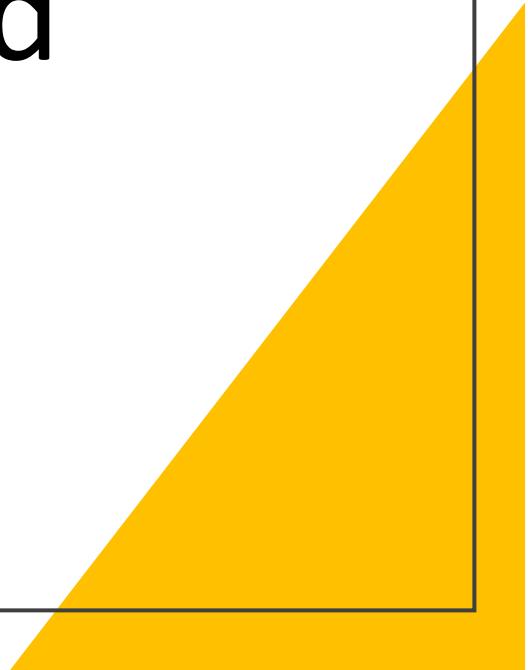
*Thank You*

# CSE 215

# Data Structures and

# Algorithms-II

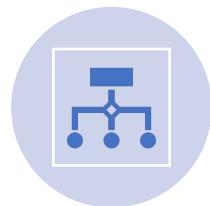
Branch and Bound



# Branch N Bound



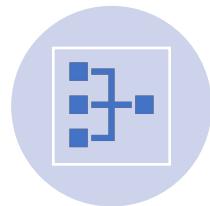
Branch and bound is an algorithm design paradigm for discrete and combinatoric optimisation problems, as well as mathematical optimisation.



A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions.



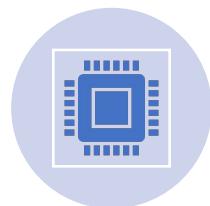
That is, the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.



The algorithm explores branches of this tree, which represent the subsets of the solution set.



Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution.



And is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

# Backtracking vs Branch N Bound



Parameter	Backtracking	Branch and Bound
Approach	Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.	Branch-and-Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution than the pre-solution leads to, it abandons that pre-solution. It completely searches the state space tree to get optimal solution.
Traversal	Backtracking traverses the state space tree by <a href="#">DFS(Depth First Search)</a> manner.	Branch-and-Bound traverse the tree in any manner, <a href="#">DFS</a> or <a href="#">BFS</a> .
Function	Backtracking involves feasibility function.	Branch-and-Bound involves a bounding function.
Problems	Backtracking is used for solving Decision Problem.	Branch-and-Bound is used for solving Optimisation Problem.
Searching	In backtracking, the state space tree is searched until the solution is obtained.	In Branch-and-Bound as the optimum solution may be present anywhere in the state space tree, so the tree need to be searched completely.
Efficiency	Backtracking is more efficient.	Branch-and-Bound is less efficient.
Applications	Useful in solving <a href="#">N-Queen Problem</a> , <a href="#">Sum of subset</a> , <a href="#">Hamilton cycle problem</a> , <a href="#">graph coloring problem</a>	Useful in solving <a href="#">Knapsack Problem</a> , <a href="#">Travelling Salesman Problem</a> .
Solve	Backtracking can solve almost any problem. (chess, sudoku, etc ).	Branch-and-Bound can not solve almost any problem.
Used for	Typically backtracking is used to solve decision problems.	Branch and bound is used to solve optimization problems.
Nodes	Nodes in state space tree are explored in depth first tree.	Nodes in tree may be explored in depth-first or breadth-first order.
Next move	Next move from current state can lead to bad choice.	Next move is always towards better solution.
Solution	On successful search of solution in state space tree, search stops.	Entire state space tree is search in order to find optimal solution.

# Branch and Bound (B & B)

---

- An enhancement of backtracking
  - **Similarity** : A state space tree is used to solve a problem.
  - **Difference** : Used only for optimization problems.
- 2 mechanisms:
  - A mechanism to generate branches when searching the solution space
  - A mechanism to generate a bound so that many branches can be terminated

# Branch and Bound

---

- Search the tree using a breadth-first search (FIFO branch and bound).
- Search the tree as in a BFS, but replace the FIFO queue with a stack (LIFO branch and bound).
- Replace the FIFO queue with a priority queue (least-cost (or max priority) branch and bound).

The priority of a node  $p$  in the queue is based on an estimate of the likelihood that the answer node is in the subtree whose root is  $p$ .

# Branch and Bound

---

- FIFO branch and bound finds solution closest to the root.
- Backtracking may never find a solution because tree depth is infinite (unless repeating configurations are eliminated).
- Least-cost branch and bound directs the search to parts of the space most likely to contain the answer. So it could perform better than backtracking.

# Branch and Bound

---

- The idea:

Set up a **bounding function**, which is used to compute a **bound** (for the value of the objective function) **at a node** on a state-space tree and determine **if it is promising**

- **Promising** (if the bound is better than the value of the best solution so far): expand beyond the node.
- **Nonpromising** (if the bound is no better than the value of the best solution so far): do not expand beyond the node (pruning the state-space tree).
- The search proceeds until all nodes have been solved or pruned.

# Bounding

---

- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
  - Greater than some number (lower bound)
  - Or, less than some number (upper bound)
- If we are looking for a maximal optimal (knapsack), then we need an upper bound
  - For example, if the best solution we have found so far has a profit of 12 and the upper bound on a node is 10 then there is no point in expanding the node
    - The node cannot lead to anything better than a 10

# Bounding

---

- We prune (via bounding) when:  
 $(currentBestSolutionCost \geq nodeBound)$
- This tells us that we get more pruning if:
  - The currentBestSolution is high
  - And the nodeBound is low
- So we want to find a high solution quickly and we want the highest possible upper bound
  - One has to factor in the extra computation cost of computing higher upper bounds vs. the expected pruning savings

# Branch and Bound

---

- It is efficient in the average case because many branches can be terminated very early.
- Although it is usually very efficient, a very large tree may be generated in the worst case.
- Many NP-hard problem can be solved by B&B efficiently in the average case; however, the worst case time complexity is still exponential.

# 0-1 Knapsack

---

- Capacity W is 10
- Upper bound is \$100  
(use fractional value)

Item	Weight	Value	Value/weight
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

# Computing Upper Bound

---

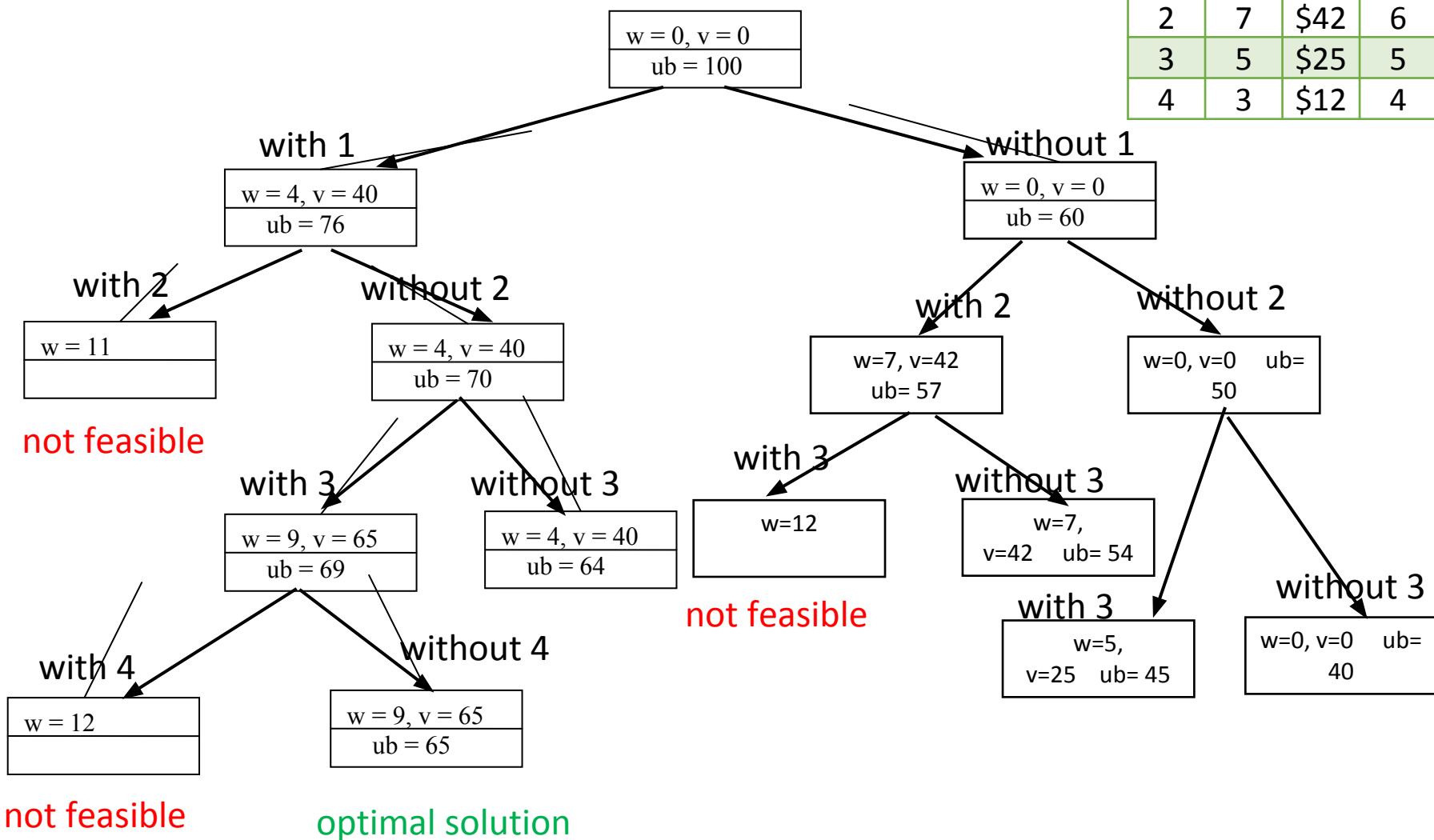
Item	Weight	Value	$V_i / W_i$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

- To compute the upper bound, we use
  - $ub = v + (W - w)(v_{i+1}/w_{i+1})$
- So the maximum upper bound is
  - pick no items, take maximum profit item
  - $ub = (10 - 0) * (\$10) = \$100$
- After we pick item 1, we calculate the upper bound as
  - all of item 1 (4, \$40) + partial of item 2 (6, \$36)
  - $\$40 + (10-4) * (\$6) = \$76$
- If we don't pick item 1:
  - $ub = (10 - 0) * (\$6) = \$60$

# State Space Tree

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$

Item	Weig ht	Valu e	Vi / Wi
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4



# Pseudocode

---

- Sort the items using Value/Weight in descending order
- Initialize Maximum Profit, **maxProfit=0**
- Create an empty queue, Q ([You can use LIFO or Priority Queue also](#))
- Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.

While Q is not empty

Do

- Extract an item from Q. Let the extracted item be u.
  - Compute profit of next level node (with the item and without the item). If the profit is more than maxProfit, then update maxProfit.
  - Compute bound( $ub = v_i + (W - w_i) * (v_{i+1} / w_{i+1})$ ) of next level node. If bound is more than maxProfit, then add next level node to Q.
- return **maxProfit**

# Travelling Salesman Problem

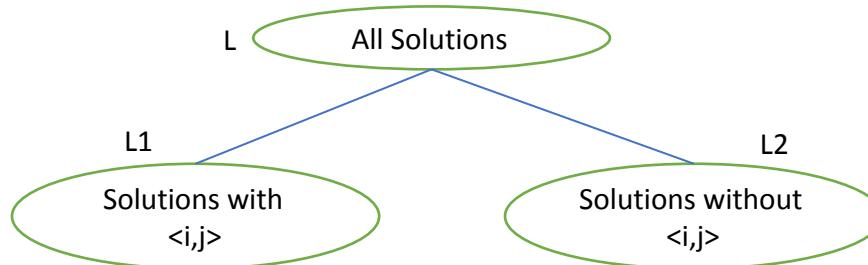
---

- Definition: Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.
- Definitions:
  - A row or column is said to be reduced iff it contains at least one zero and all remaining entries are non-negative.
  - A matrix is reduced iff every row and column is reduced.

# Branching

- **Branching:**

- Each node splits the remaining solutions into two groups: those that include a particular edge and those that exclude that edge.
- Each node has a lower bound.
- Example: Given a graph  $G=(V,E)$ , let  $\langle i,j \rangle \in E$ ,



# Bounding

How to compute the cost of each node?

- Subtract of a constant from any row and any column does not change the optimal solution (The path).
- The cost of the path changes but the path itself.
- Let A be the cost matrix of a  $G=(V,E)$ .
- The cost of each node in the search tree is computed as follows:
  - Let R be a node in the tree and  $A^R$  its reduced matrix
  - The cost of the child  $R^S$ :
    - Set row i and column j to infinity
    - Set  $A(j,i)$  to infinity
    - Reduced S and let RCL be the reduced cost.
    - $C(S) = C^R + RCL + A(i,j)$

# Bounding cont..



**Get the reduced matrix  $A'$  of  $A$  and let  $L$  be value subtracted from  $A$**



**$L$ : represents the lower bound of the path solution**



**The cost of the path is exactly reduced by  $L$ .**



**What to determine the branching edge?**

The rule favors a solution through left subtree rather than right subtree, i.e. the matrix is reduced by a dimension.

Note that the right subtree only sets the branching edge to infinity.

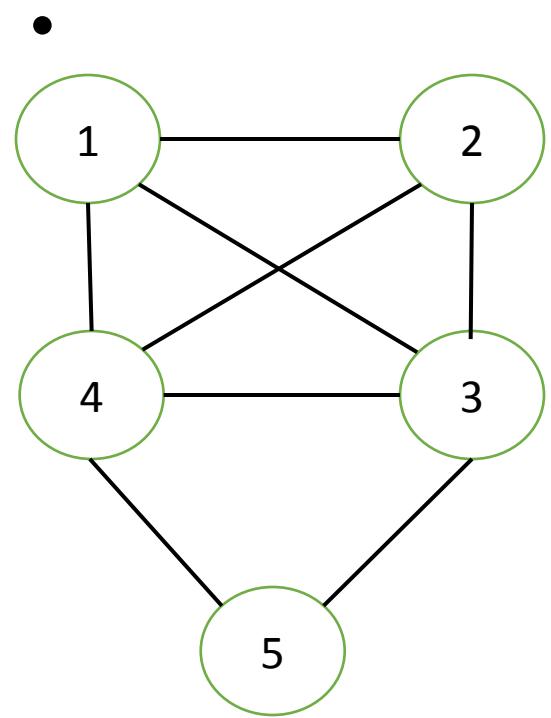
Pick the edge that causes the greatest increase in the lower bound of the right subtree, i.e. the lower bound of the root of the right subtree is greater.

# Cost Determine

Example:

- The reduced cost matrix is done as follows:
  - Change all entries of row  $i$  and column  $j$  to infinity
  - Set  $A(j,1)$  to infinity (assuming the start node is 1)
  - Reduce all rows first and then column of the resulting matrix

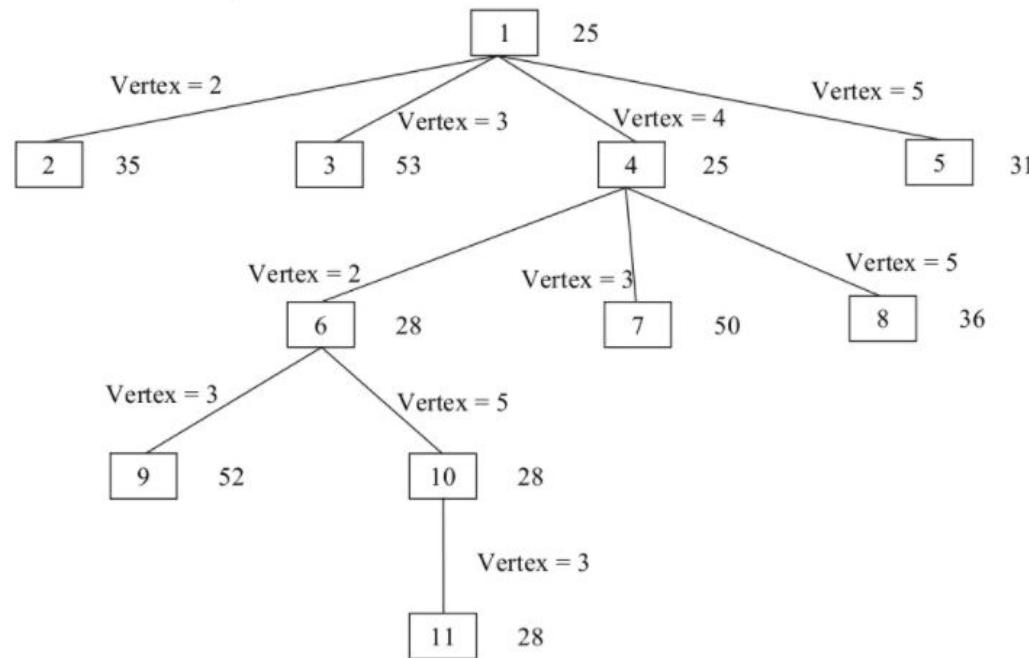
# TSP



Cost matrix

inf	20	30	10	11
15	inf	16	4	2
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

# State Space Tree



- The TSP starts from node 1: **Node 1**

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #1: reduce by 10

Before

inf	20	30	10	11
15	inf	16	4	2
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

inf	10	20	0	1
15	inf	16	4	2
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #2: reduce by 2

Before

inf	10	20	0	1
15	inf	16	4	2
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

inf	10	20	0	1
13	inf	14	2	0
3	5	inf	2	4
19	6	18	inf	3
16	4	7	16	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #3: reduce by 2

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 3 & 5 & \text{inf} & 2 & 4 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 13 & \text{inf} & 14 & 2 & 0 \\ 1 & 3 & \text{inf} & 0 & 2 \\ 19 & 6 & 18 & \text{inf} & 3 \\ 16 & 4 & 7 & 16 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #4: reduce by 3

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
19	6	18	inf	3
16	4	7	16	inf

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
16	4	7	16	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Row #5: reduce by 4

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
16	4	7	16	inf

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
12	0	3	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#1: reduce by 1

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
12	0	3	12	inf

inf	10	20	0	1
12	inf	14	2	0
0	3	inf	0	2
15	3	15	inf	0
11	0	3	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#2: It is reduced (**no change**)

Before

inf	10	20	0	1
13	inf	14	2	0
1	3	inf	0	2
16	3	15	inf	0
12	0	3	12	inf

inf	10	20	0	1
12	inf	14	2	0
0	3	inf	0	2
15	3	15	inf	0
11	0	3	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#3: Reduce by 3

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 12 & \text{inf} & 14 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 15 & \text{inf} & 0 \\ 11 & 0 & 3 & 12 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#4: It is reduced (**no change**)

Before

inf	10	20	0	1
12	inf	14	2	0
0	3	inf	0	2
15	3	15	inf	0
11	0	3	12	inf

inf	10	17	0	1
12	inf	11	2	0
0	3	inf	0	2
15	3	12	inf	0
11	0	0	12	inf

After

- The TSP starts from node 1: Node 1

- Reduced Matrix: to get the lower bound of the path starting at node 1
  - Column#5: It is reduced (no change)

Before

$$\begin{bmatrix} \text{inf} & 10 & 20 & 0 & 1 \\ 12 & \text{inf} & 14 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 15 & \text{inf} & 0 \\ 11 & 0 & 3 & 12 & \text{inf} \end{bmatrix}$$
$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

After

The reduced matrix is:

Cost (1) = 25

$$\begin{bmatrix} \text{inf} & 10 & 17 & 0 & 1 \\ 12 & \text{inf} & 11 & 2 & 0 \\ 0 & 3 & \text{inf} & 0 & 2 \\ 15 & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 2: **Node 2**
  - Cost of edge  $\langle 1,2 \rangle$  is:  $A(1,2) = 10$
  - Set row#1 = **inf** since we are choosing edge  $\langle 1,2 \rangle$
  - Set column #2= **inf** since we are choosing edge  $\langle 1,2 \rangle$
  - Set  $A(2,1) = \text{inf}$

**The resulting cost matrix is:**

The matrix is reduced:

RCL=0 (As min is 0 for all row

And columns)

The cost of node 2 (Considering Vertex 2 from vertex 1) is:

Cost(2)

$$= \text{cost}(1) + \text{RCL} + A(1,2) = 25 + 0 + 10$$

$$= 35$$

inf	inf	inf	inf	inf
inf	inf	11	2	0
0	inf	inf	0	2
15	inf	12	inf	0
11	inf	0	12	inf

- Choose to go to vertex 3: **Node 3**

- Cost of edge  $\langle 1,3 \rangle$  is:  $A(1,2) = 17$  (in the reduced matrix)
- Set row#1 = **inf** since we are staring from node 1
- Set column #3= **inf** since we are choosing edge  $\langle 1,3 \rangle$
- Set  $A(3,1) = \text{inf}$

The resulting cost matrix is:

Reduce the matrix:

- Rows are reduced
- The colum are reduced except for column # 1:
  - Reduce column 1 by 11

inf	inf	inf	inf	inf
12	inf	inf	2	0
inf	3	inf	0	2
15	3	inf	inf	0
11	0	inf	12	inf

- The lower bound is:

- RCL=11

- The cost of going through node 3 is:

- $$\begin{aligned} \text{Cost}(3) &= \text{cost}(1) + \text{RCL} + A(1,3) = 25+11+17 \\ &= 53 \end{aligned}$$

Before

inf	inf	inf	inf	inf
12	inf	inf	2	0
inf	3	inf	0	2
15	3	inf	inf	0
11	0	inf	12	inf

inf	inf	inf	inf	inf
1	inf	inf	2	0
inf	3	inf	0	2
4	3	inf	inf	0
0	0	inf	12	inf

After

- Choose to go to vertex 4: **Node 4**
  - Remember that the cost matrix is the one that was reduced at the starting vertex 1
    - Cost of edge  $\langle 1,4 \rangle$  is:  $A(1,4) = 0$
    - Set row#1 = **inf** since we are staring from node 1
    - Set column #4= **inf** since we are choosing edge  $\langle 1,4 \rangle$
    - Set  $A(4,1) = \text{inf}$
- The resulting cost matrix is:**

Reduce the matrix:

- Rows are reduced
- Columns are reduced

- The cost of going through node 4 is:
  - $\text{Cost}(4) = \text{cost}(1) + \text{RCL} + A(1,4)$
  - $= 25+0+0$
  - $= 25$

inf	inf	inf	inf	inf
12	inf	11	inf	0
0	3	inf	inf	2
inf	3	12	inf	0
11	0	0	inf	inf

- Choose to go to vertex 5: **Node 5**
  - Remember that the cost matrix is the one that was reduced at the starting vertex 1
    - Cost of edge  $\langle 1,5 \rangle$  is:  $A(1,5) = 1$
    - Set row#1 = **inf** since we are staring from node 1
    - Set column #5= **inf** since we are choosing edge  $\langle 1,5 \rangle$
    - Set  $A(5,1) = \text{inf}$
- The resulting cost matrix is:**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & 2 & \text{inf} \\ 0 & 3 & \text{inf} & 0 & \text{inf} \\ 15 & 3 & 12 & \text{inf} & \text{inf} \\ \text{inf} & 0 & 0 & 12 & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
    - Reduce Row#2: by 2
    - Reduce Row# 4: by 3
  - Columns are reduced
- The lower bound is:
  - $RCL = 2+3 = 5$
- The cost of going through node 5 is:
  - $\text{Cost}(5) = \text{cost}(1) + RCL + A(1,5)$   
 $= 25+5+1$   
 $= 31$

Reduce by 2

inf	inf	inf	inf	inf
10	inf	9	0	inf
0	3	inf	0	inf
15	3	12	inf	inf
inf	0	0	12	inf

Reduce by 3

inf	inf	inf	inf	inf
10	inf	9	0	inf
0	3	inf	0	inf
12	0	9	inf	inf
inf	0	0	12	inf

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 → 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 → 3
  - 4:  $\text{cost}(4) = 25$ , path: 1 → 4
  - 5:  $\text{cost}(5) = 31$ , path: 1 → 5
- Explore the node with the lowest cost: Node 4 has a cost of 25
- Vertices to be explored from node 4: 2, 3, and 5

- Now we are starting from the cost matrix at node 4 is:

$$\text{Cost}(4) = 25$$
$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & \text{inf} & 0 \\ 0 & 3 & \text{inf} & \text{inf} & 2 \\ \text{inf} & 3 & 12 & \text{inf} & 0 \\ 11 & 0 & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 2: **Node 6** (path 1→4→2)
  - Cost of edge <4,2> is:  $A(4,2) = 3$
  - Set row#4 = **inf** since we are considering edge <4,2>
  - Set column # 2= **inf** since we are considering edge <4,2>
  - Set  $A(2,1) = \text{inf}$

- **Reduce the matrix:**

- Rows are reduced
- Columns are reduced
- The lower bound is  $RCL = 0$
- The cost of going through node 2 is:

$$\text{Cost}(6) = \text{cost}(4) + RCL + A(4,2)$$

$$= 25 + 0 + 3$$

$$= 28$$

inf	inf	inf	inf	inf
inf	inf	11	inf	0
0	inf	inf	inf	2
inf	inf	inf	inf	inf
11	inf	0	inf	inf

- Choose to go to vertex 3: **Node 7** (path 1→4→3)
  - Cost of edge <4,3> is:  $A(4,3) = 12$
  - Set row#4 = **inf** since we are considering edge <4,3>
  - Set column # 3= **inf** since we are considering edge <4,3>
  - Set  $A(3,1) = \text{inf}$

The resulting matrix is

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & \text{inf} & \text{inf} & 0 \\ \text{inf} & 3 & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & 0 & \text{inf} & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
  - Reduce Row#3:
  - Reduce column#1:
- The lower bound is:
  - $RCL = 2+11 = 13$
- Considering vertex 3 from vertex 4 is:
  - $\text{Cost}(7) = \text{cost}(4) + RCL + A(4,3)$   
 $= 25+13+12$   
 $= 50$

Reduce by 2

inf	inf	inf	inf	inf
12	inf	inf	inf	0
inf	1	inf	inf	0
inf	inf	inf	inf	inf
11	0	inf	inf	inf

inf	inf	inf	inf	inf
1	inf	inf	inf	0
inf	1	inf	inf	0
inf	inf	inf	inf	inf
0	0	inf	inf	inf

Reduce by 11

- Choose to go to vertex 5: **Node 8** (path 1→4→5)
  - Cost of edge <4,5> is:  $A(4,5) = 0$
  - Set row#4 = **inf** since we are considering edge <4,3>
  - Set column # 3= **inf** since we are considering edge <4,5>
  - Set  $A(5,1) = \text{inf}$

The resulting matrix is

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 12 & \text{inf} & 11 & \text{inf} & \text{inf} \\ 0 & 3 & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & 0 & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
    - Reduce Row#2
  - Columns are reduced
- The lower bound is:
  - $RCL = 11 = 11$
- Considering vertex 5 from vertex 4 is:
  - $\text{Cost}(8) = \text{cost}(4) + RCL + A(4,5)$   
 $= 25 + 11 + 0$   
 $= 36$

Reduce by 11

inf	inf	inf	inf	inf
1	inf	0	inf	inf
0	3	inf	inf	inf
inf	inf	inf	inf	inf
inf	0	0	inf	inf

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 ↗ 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 ↗ 3
  - 5:  $\text{cost}(5) = 31$ , path: 1 ↗ 5
  - 6:  $\text{cost}(6) = 28$ , path: 1 ↗ 4 ↗ 2
  - 7:  $\text{cost}(7) = 50$ , path: 1 ↗ 4 ↗ 3
  - 8:  $\text{cost}(8) = 36$ , path: 1 ↗ 4 ↗ 5
- Explore the node with the lowest cost: Node 6 has a cost of 28
- Vertices to be explored from node 6: 3, and 5

- Now we are starting from the cost matrix at node 6 is:

**Cost(4) =28**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & 11 & \text{inf} & 0 \\ 0 & \text{inf} & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & \text{inf} & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 3: **Node 9** (path 1→4→2→3)

- Cost of edge <2,3> is:  $A(2,3) = 11$
- Set row#2 = **inf** since we are considering edge <2,3>
- Set column # 3= **inf** since we are considering edge <2,3>
- Set  $A(3,1) = \text{inf}$

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & 2 \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 11 & \text{inf} & \text{inf} & \text{inf} & \text{inf} \end{bmatrix}$$

Reduce the matrix:

- Reduce Rows:
  - Reduce Row#3
  - Reduce column# 1

• The lower bound is:

- $RCL = 2 + 11 = 13$

• Considering vertex 3 from vertex

2 is:

- $\text{Cost}(9) = \text{cost}(6) + RCL + A(2,3)$   
 $= 28 + 13 + 11$   
 $= 52$

By 2

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	0
inf	inf	inf	inf	inf
11	inf	inf	inf	inf

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	0
inf	inf	inf	inf	inf
0	inf	inf	inf	inf

- Choose to go to vertex 5: **Node 10** (path 1→4→2→5)
  - Cost of edge <2,5> is:  $A(2,5) = 0$
  - Set row#2 = **inf** since we are considering edge <2,3>
  - Set column # 3= **inf** since we are considering edge <2,3>
  - Set  $A(5,1) = \text{inf}$

- **Reduce the matrix:**

- Rows are reduced
- Columns are reduced

- The lower bound is  $RCL = 0$
- Considering vertex 3 from vertex 2 is:

$$\text{Cost}(10) = \text{cost}(6) + RCL + A(2,3)$$

$$= 28 + 0 + 0 \\ = 28$$

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
0	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	0	inf	inf

# In summary

- So the live nodes we have so far are:
  - 2:  $\text{cost}(2) = 35$ , path: 1 ⊕ 2
  - 3:  $\text{cost}(3) = 53$ , path: 1 ⊕ 3
  - 5:  $\text{cost}(5) = 31$ , path: 1 ⊕ 5
  - 7:  $\text{cost}(7) = 50$ , path: 1 ⊕ 4 ⊕ 3
  - 8:  $\text{cost}(8) = 36$ , path: 1 ⊕ 4 ⊕ 5
  - 9:  $\text{cost}(9) = 52$ , path: 1 ⊕ 4 ⊕ 2 ⊕ 3
  - 10:  $\text{cost}(2) = 28$ , path: 1 ⊕ 4 ⊕ 2 ⊕ 5
- Explore the node with the lowest cost: Node 10 has a cost of 28
- Vertices to be explored from node 10: 3

- Now we are starting from the cost matrix at node 10 is:

**Cost(4) =28**

$$\begin{bmatrix} \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ 0 & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & \text{inf} & \text{inf} & \text{inf} \\ \text{inf} & \text{inf} & 0 & \text{inf} & \text{inf} \end{bmatrix}$$

- Choose to go to vertex 3: **Node 11** (path 1?4?2?5?3)

- Cost of edge  $\langle 5,3 \rangle$  is:  $A(5,3) = 0$
- Set row#5= **inf** since we are considering edge  $\langle 5,3 \rangle$
- Set column # 3= **inf** since we are considering edge  $\langle 5,3 \rangle$
- Set  $A(3,1) = \text{inf}$

- **Reduce the matrix:**

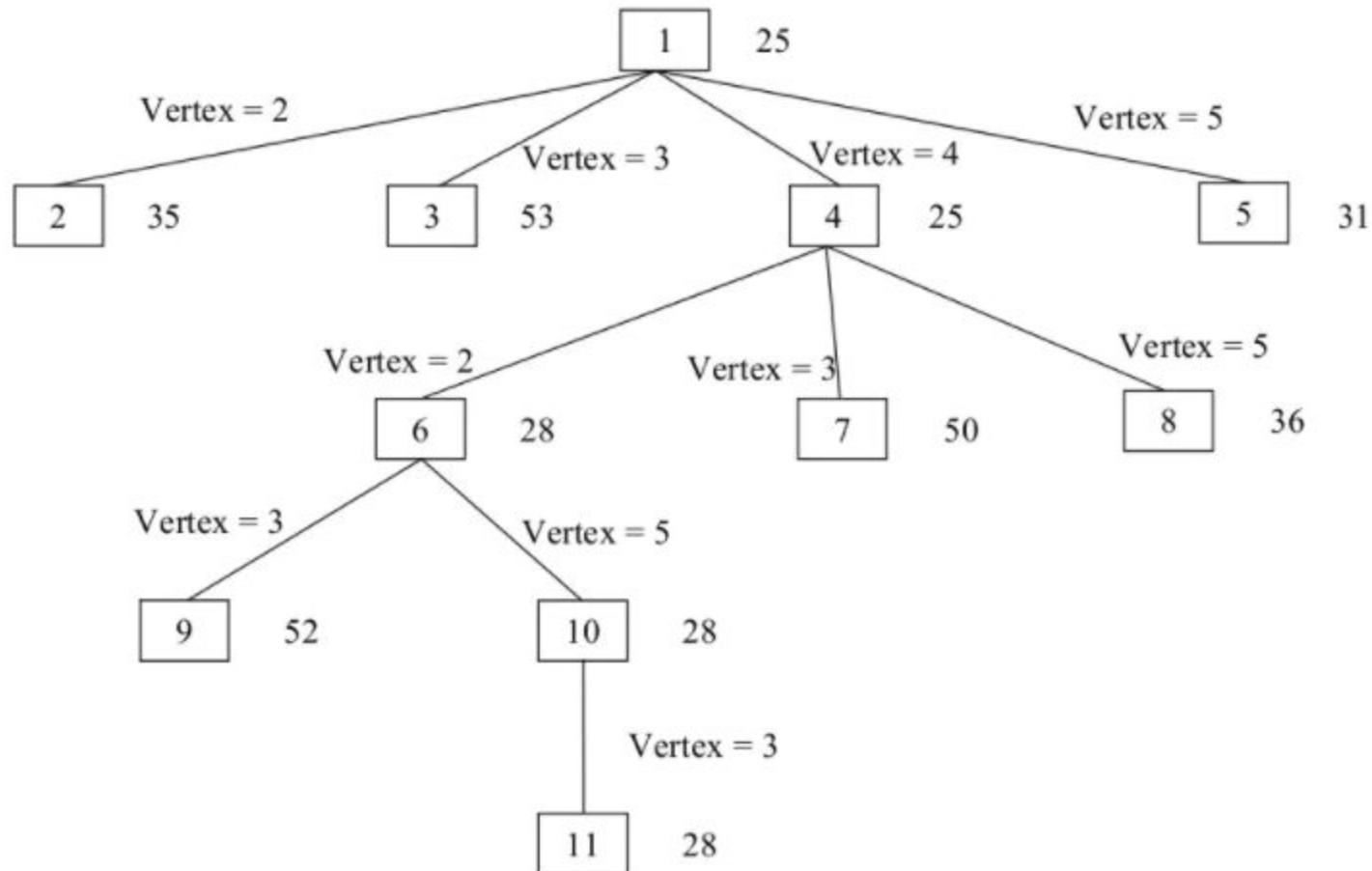
- Rows are reduced
- Columns are reduced

- The lower bound is  $RCL = 0$
- Considering vertex 5 from vertex 3 is:

$$\begin{aligned} \text{Cost}(11) &= \text{cost}(10) + RCL + A(5,3) \\ &= 28 + 0 + 0 \\ &= 28 \end{aligned}$$

inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	inf
inf	inf	inf	inf	inf

# State Space Tree



The short tour: path 1 → 4 → 2 → 5 → 3 → 1)

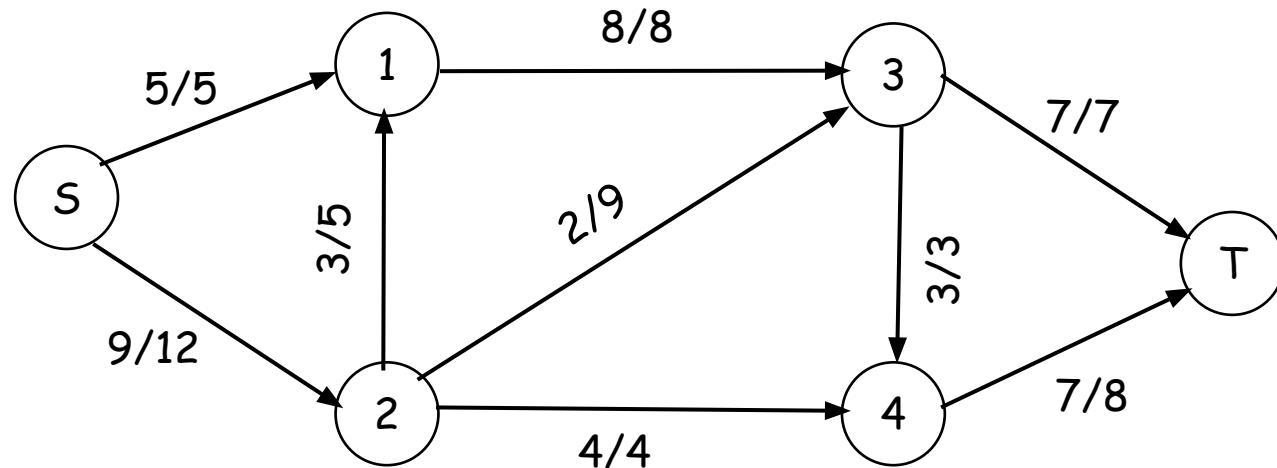
*Thank You*

# **Graph Algorithms: Maximum Flow**

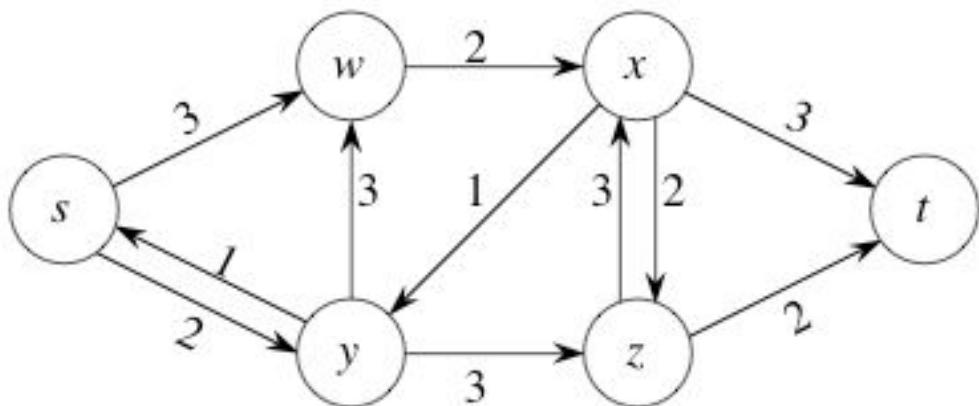


# Flow Network Applications

- Liquids flowing through pipes
- Parts through assembly lines
- Current through electrical networks
- Information through communication networks



# Flow Network



- Directed graph  $G = (V, E)$  with non-negative edge weights  $c : E \rightarrow R$ 
  - $c(u, v)$ : nonnegative *capacity* of an edge  $(u, v) \in E$ 
    - $c(u, v) = 0$  if  $(u, v) \notin E$
  - $s$ : source of the network
  - $t$ : sink of the network

# Flow Network

- A *positive flow* is a function  $f: V \times V \rightarrow \mathbb{R}$  s.t.,

- Capacity constraint:

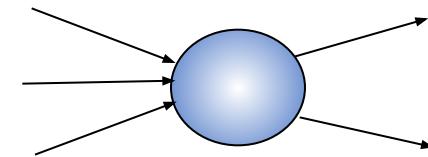
- For all  $u, v \in V$ ,  $0 \leq f(u, v) \leq c(u, v)$

- Skew Symmetry*:

- For all  $u, v \in V$ ,  $f(u, v) = -f(v, u)$

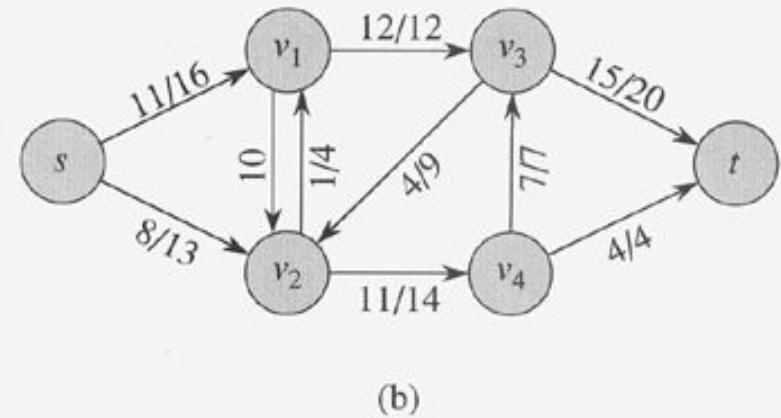
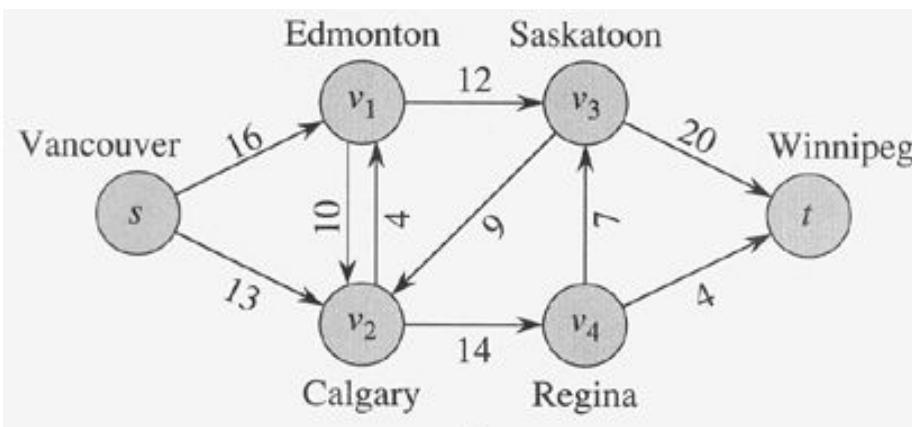
- Flow conservation constraint:

- For all  $u \in V - \{s, t\}$ ,



$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

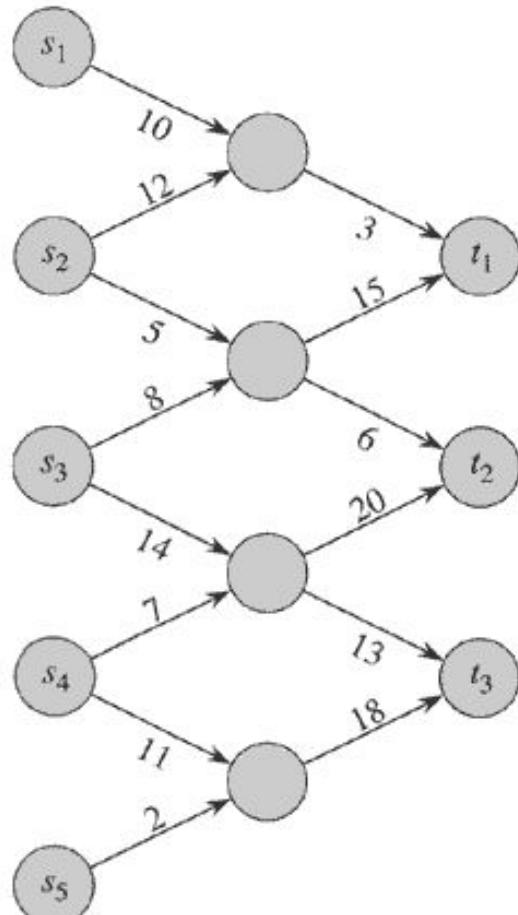
Flow in equals flow out



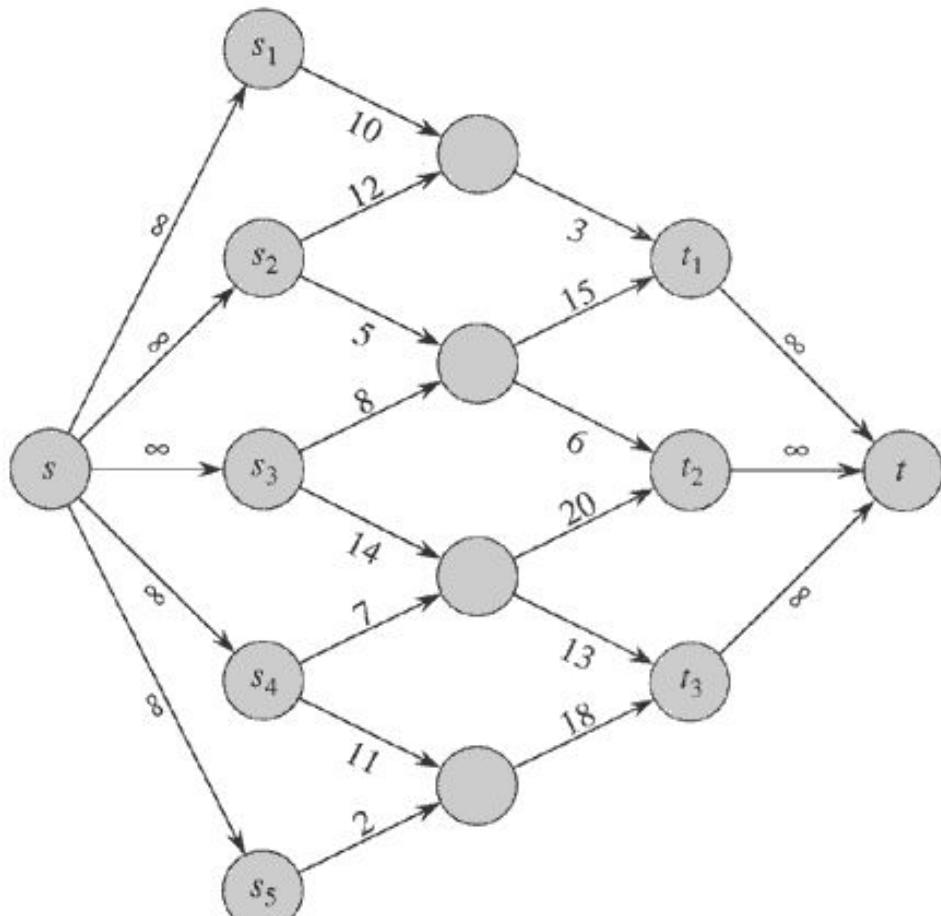
# Maximum-Flow Problem

- Given a flow  $f$ , the value of  $f$  is
  - $|f| = \sum f(s, u)$  : i.e., total flow out of the source
- Maximum-flow problem:
  - Compute a flow of maximum value
- Multiple sources/sinks
  - Convert to single source/sink problem by adding one supersource and one supersink
- Anti-parallel edges
  - Transform the network into an equivalent one containing no anti-parallel edges by adding a new vertex

# Maximum-Flow Problem



(a)



(b)

# Maximum-Flow Problem

FORD-FULKERSON-METHOD( $G, s, t$ )

- 1 initialize flow  $f$  to 0
- 2 **while** there exists an augmenting path  $p$
- 3     **do** augment flow  $f$  along  $p$
- 4 **return**  $f$

# Residual Network

- Given a flow network and a flow, the **residual network** consists of edges that can admit more network flow.
- $G = (V, E)$ : a flow network with source  $s$  and sink  $t$
- $f$ : a flow in  $G$
- The amount of additional network flow from  $u$  to  $v$  before exceeding the capacity  $c(u, v)$  is the **residual capacity** of  $(u, v)$ , given by:  $c_f(u, v) = c(u, v) - f(u, v)$

# Residual Network

- Given a flow  $f$  in a network  $G = (V, E)$

- The residual capacity between  $u, v \in V$

$$c_f(u, v) = c(u, v) - f(u, v) \geq 0 !$$

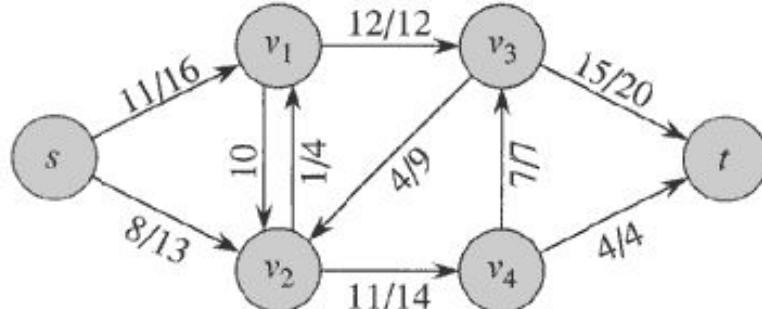
- Residual network  $G_f = (V, E_f)$
- where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$

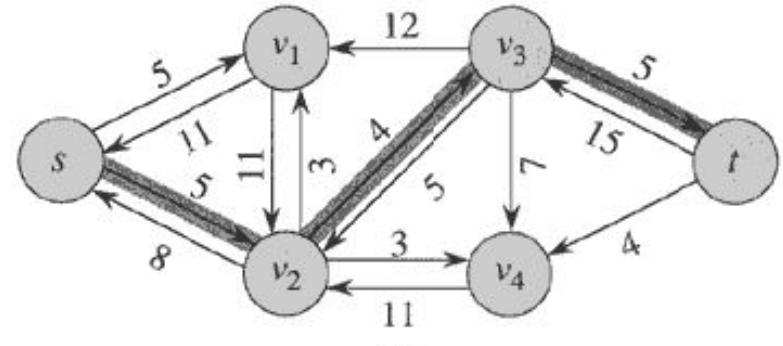
- Residual network  $G_f$  may also contain edges that are not in  $G$
- Residual capacity,  $c_f(u, v)$  is defined by

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (u, v) \notin E \\ 0 & \text{otherwise} \end{cases}$$

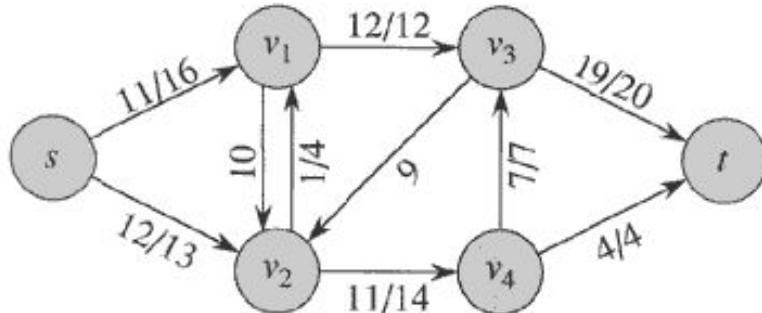
# Residual Network: Example



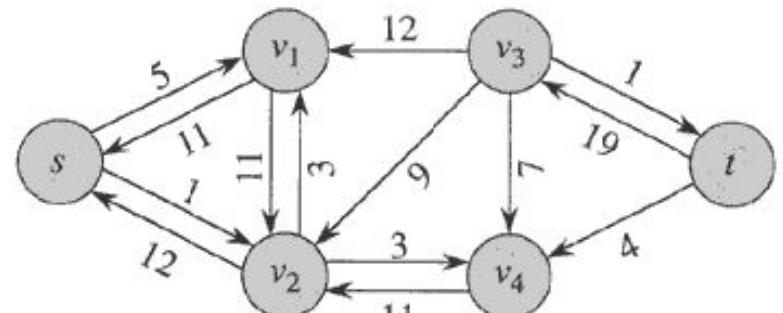
(a)



(b)



(c)



(d)

Each edge in  $G$  corresponds to at most two edges in residual network:  $|E_f| \leq 2 |E|$

# Residual Network

- A flow in a residual network provides a roadmap for adding flow to the original flow network.
  - If  $f$  is a flow in  $G$  and  $f'$  is a flow in  $G_f$ , we define  $f \uparrow f'$ , the *augmentation of flow  $f$  by  $f'$*

$$(f \uparrow f')(u, v) = \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

- Pushing flow on the reverse edge in the residual network is known as *cancellation*.

# Residual Network

- Lemma

- Let  $G = (V, E)$  be a flow network with source  $s$  and sink  $t$ , and let  $f$  be a flow in  $G$ .
- Let  $G_f$  be the residual network of  $G$  induced by  $f$ , and let  $f'$  be a flow in  $G_f$ . Then the flow sum  $f + f'$  is a flow in  $G$  with value  $|f + f'| = |f| + |f'|$
- $f + f'$ : the flow in the same direction will be added.  
the flow in different directions will be cancelled.

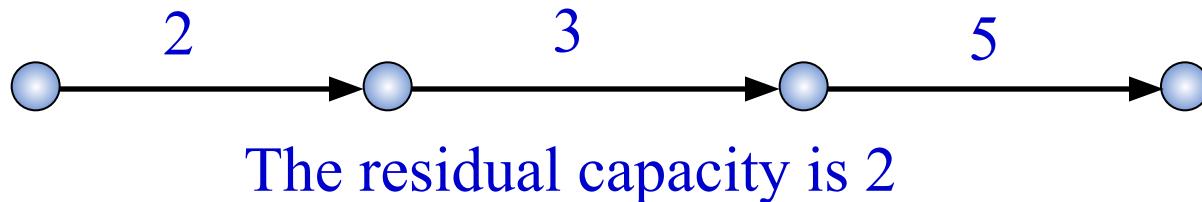
This suggests that we can improve current flow by computing a new flow for its residual network, and add it upon original one

# Augmenting Path

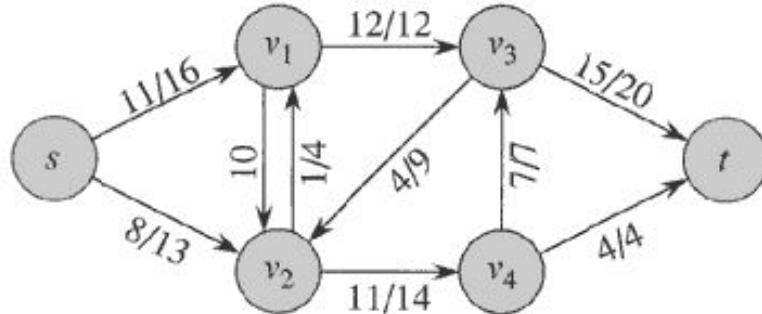
- Given a flow  $f$  in a flow network  $G = (V, E)$ , an *augmenting path*  $p$  is a simple path from source  $s$  to sink  $t$  in the residual network  $G_f$
- How much extra flow can we push on an augmenting path  $p$  ?

The maximum amount by which we increase the flow on each edge in an augmenting path  $p$  is the *residual capacity* of  $p$ , given by

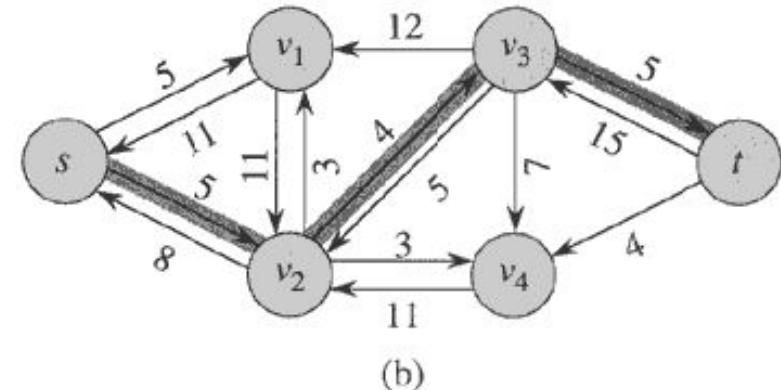
$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$$



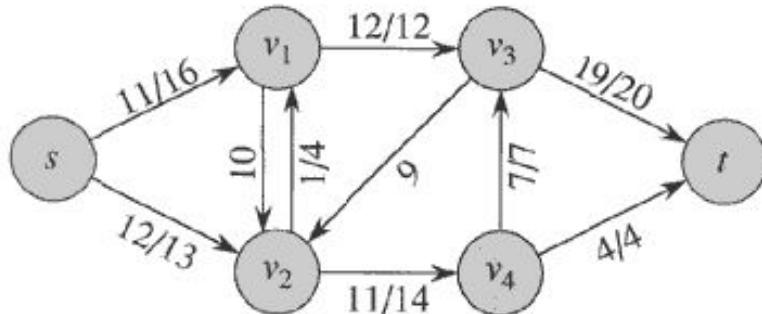
# Augmenting Path: Example



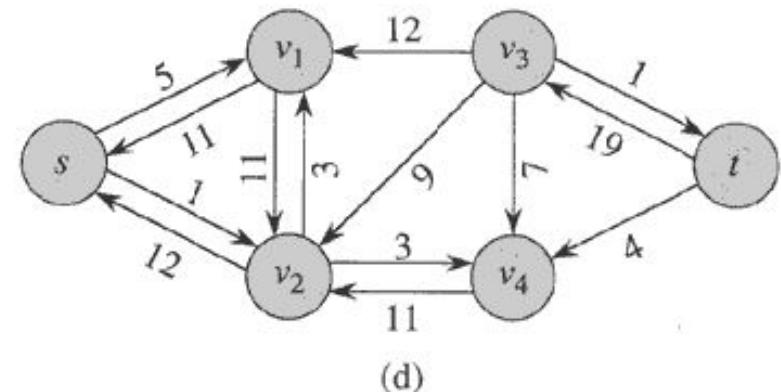
(a)



(b)



(c)



(d)

# Lemma: Augmenting $\rightarrow$ Flow

*Lemma:*

Given flow network  $G$ , flow  $f$  in  $G$ , residual network  $G_f$ . Let  $p$  be an augmenting path in  $G_f$ . Define  $f_p : V \times V \rightarrow \mathbf{R}$ :

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p , \\ -c_f(p) & \text{if } (v, u) \text{ is on } p , \\ 0 & \text{otherwise .} \end{cases}$$

Then  $f_p$  is a flow in  $G_f$  with value  $|f_p| = c_f(p) > 0$ .

*Corollary:*

Given flow network  $G$ , flow  $f$  in  $G$ , and an augmenting path  $p$  in  $G_f$ , define  $f_p$  as in lemma, and define  $f' : V \times V \rightarrow \mathbf{R}$  by  $f' = f + f_p$ . Then  $f'$  is a flow in  $G$  with value  $|f'| = |f| + c_f(p) > |f|$ .

# Ford-Fulkerson Algorithm

**FORD-FULKERSON( $G, s, t$ )**

for each edge  $(u, v) \in E[G]$  do

$$\left. \begin{array}{l} f[u, v] = 0 \\ f[v, u] = 0 \end{array} \right\} O(E)$$

<-- while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$  do -->

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is in } p\}$$

for each edge  $(u, v)$  in  $p$  do

if  $(u, v) \in E[G]$

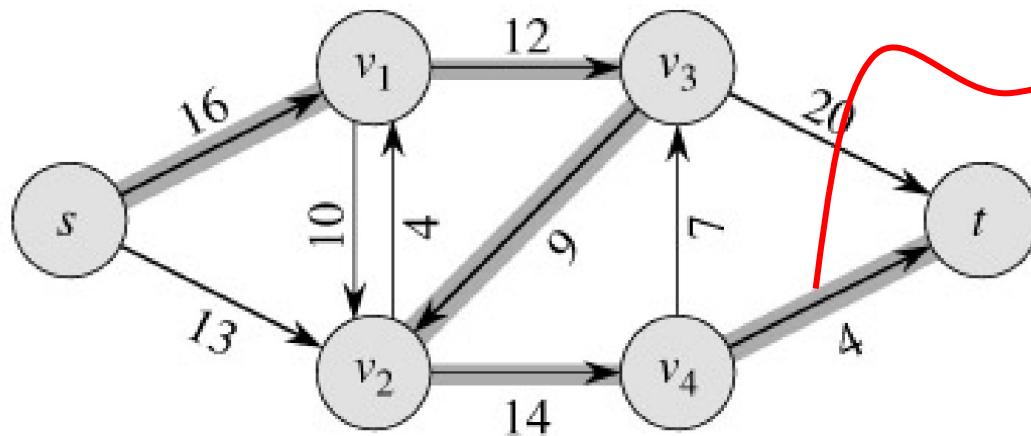
$$f[u, v] = f[u, v] + c_f(p)$$

else  $f[v, u] = f[v, u] - c_f(p)$

$\left. \begin{array}{l} \\ \\ \end{array} \right\} O(E)$

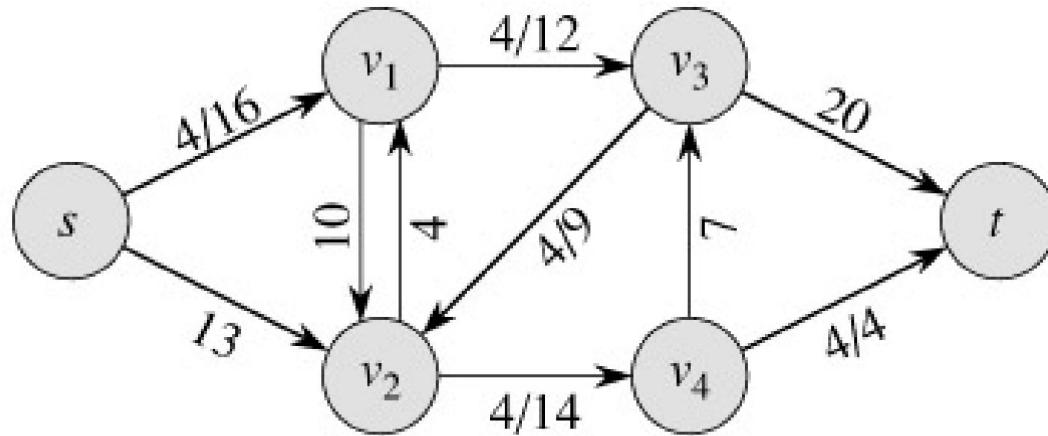
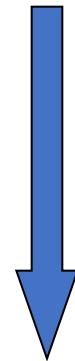
?

# Ford-Fulkerson Algorithm: Example



augmenting path

Original Network

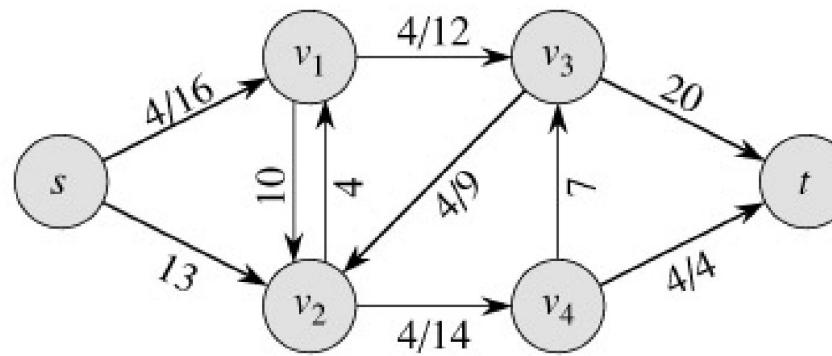


Flow Network

Resulting Flow = 4

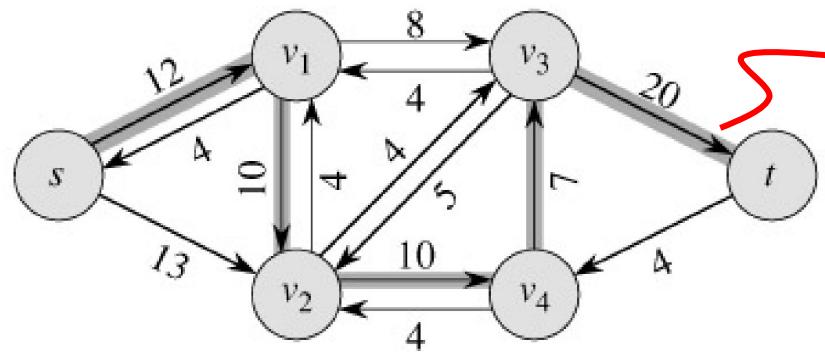
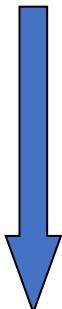
# Ford-Fulkerson Algorithm: Example

Flow Network



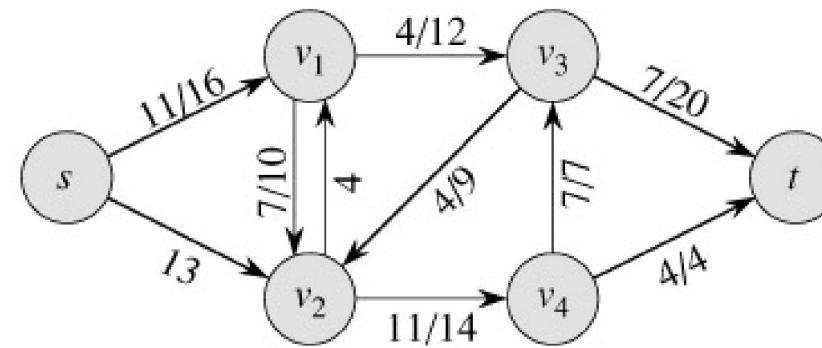
Resulting Flow = 4

Residual Network



augmenting path

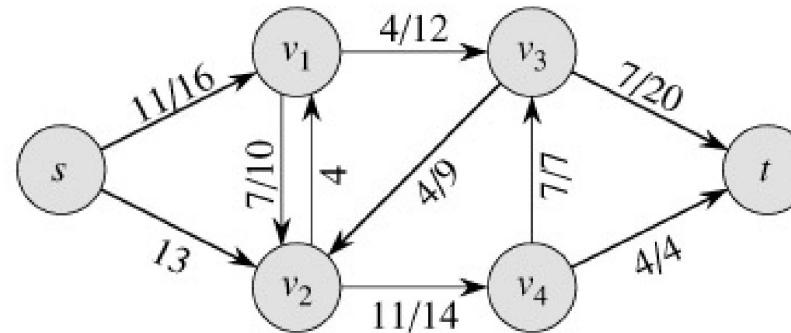
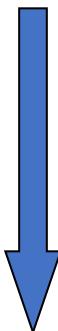
Flow Network



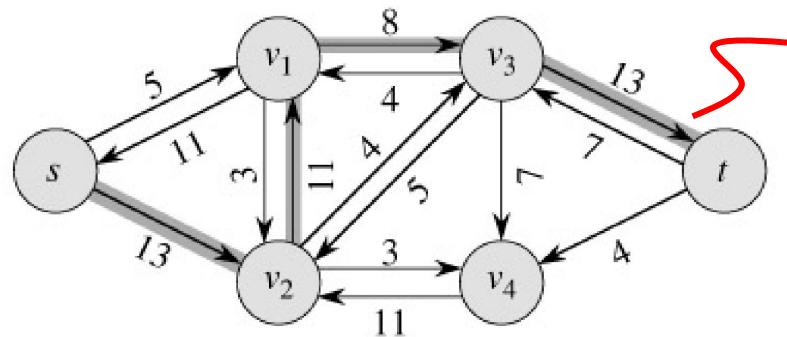
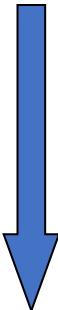
Resulting Flow = 11

# Ford-Fulkerson Algorithm: Example

Flow Network

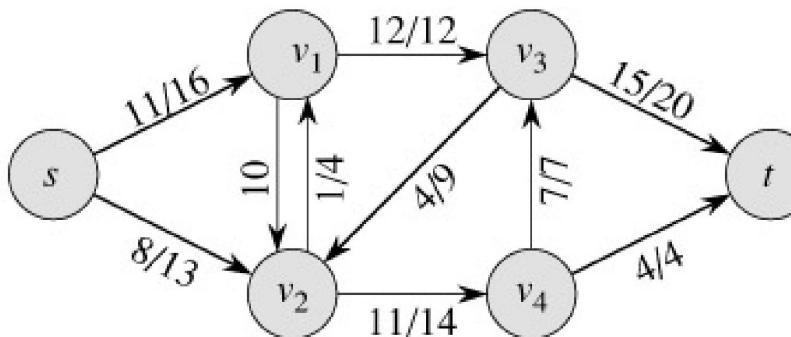


Residual Network



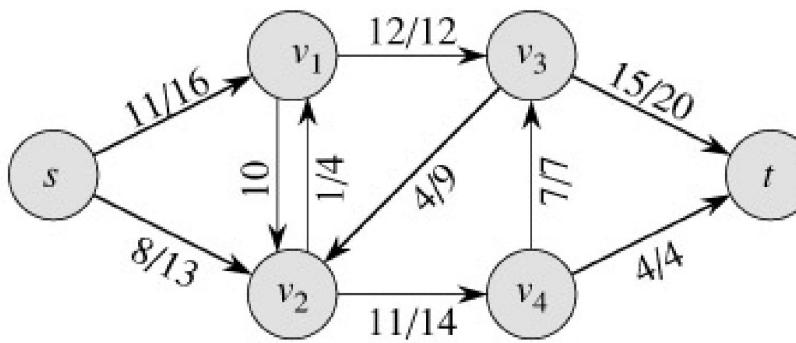
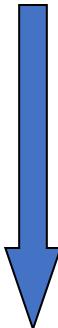
Flow Network

Resulting Flow = 19



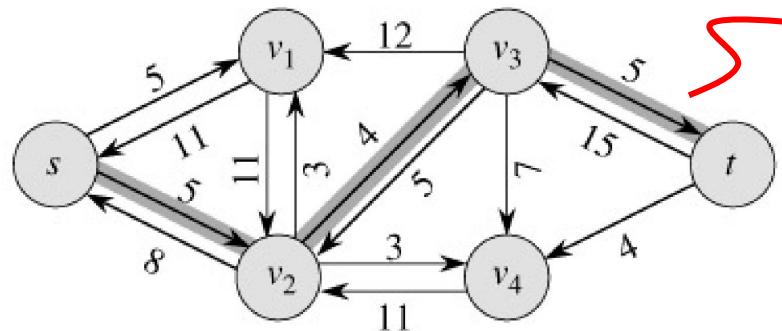
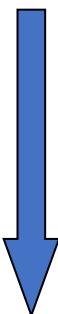
# Ford-Fulkerson Algorithm: Example

Flow Network



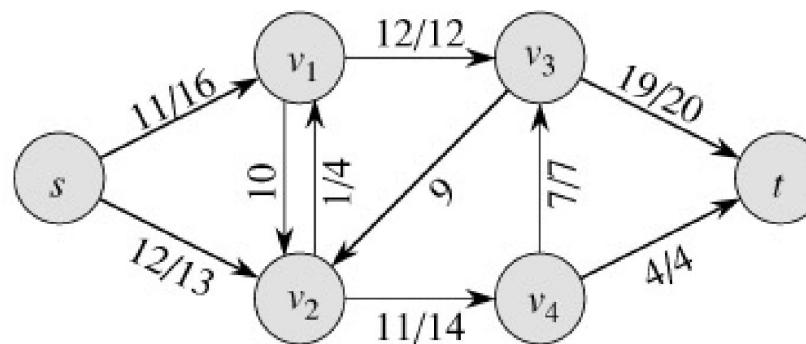
Resulting Flow = 19

Residual Network



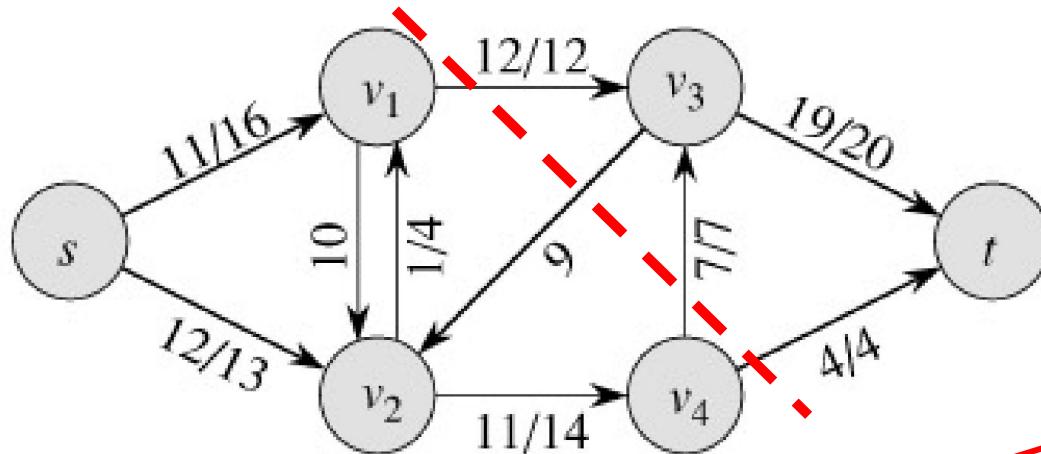
augmenting path

Flow Network



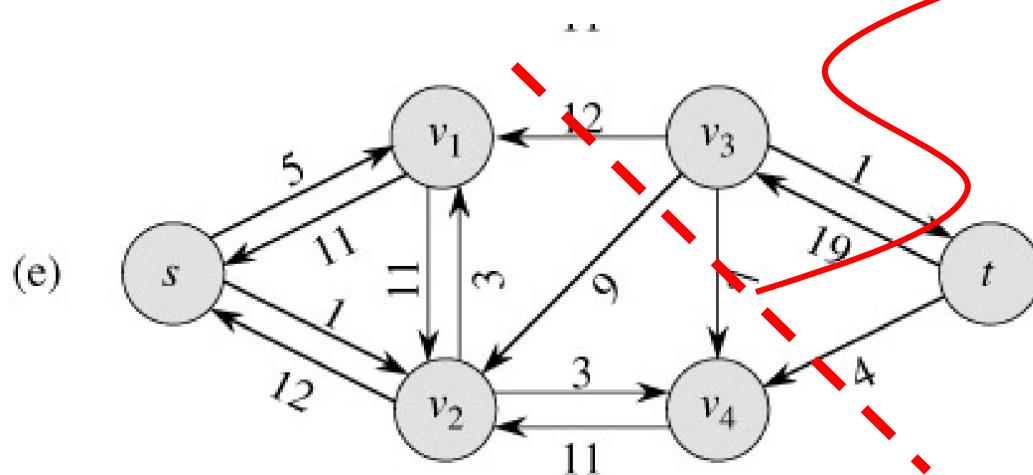
Resulting Flow = 23

# Ford-Fulkerson Algorithm: Example



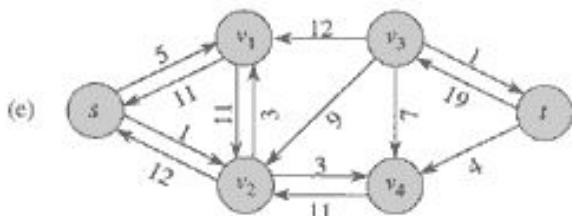
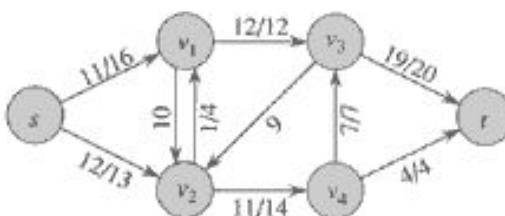
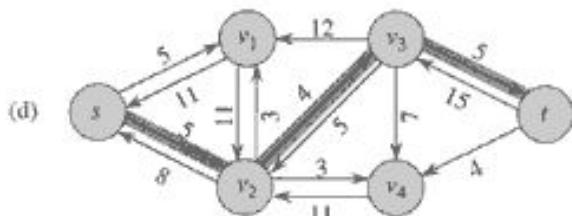
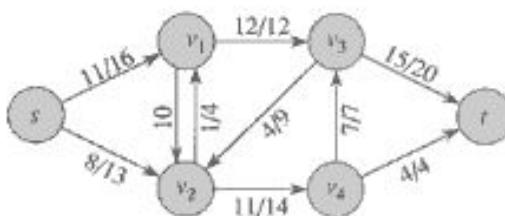
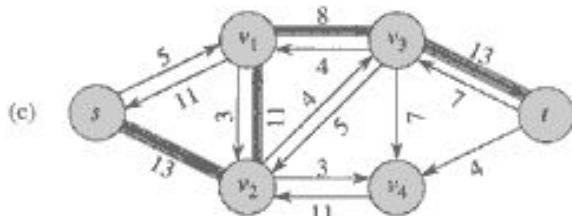
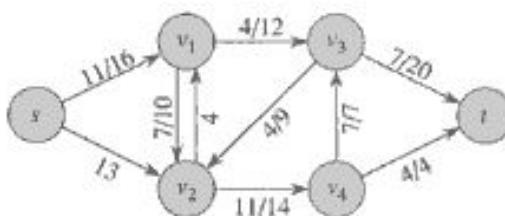
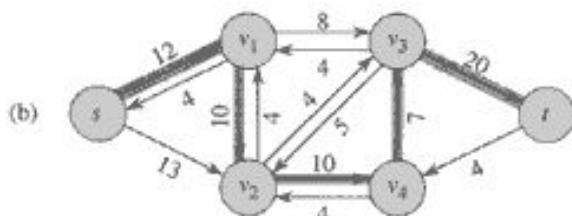
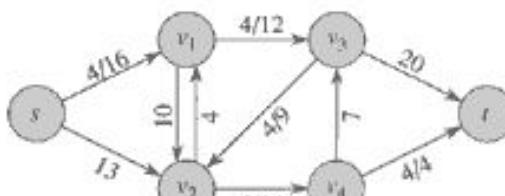
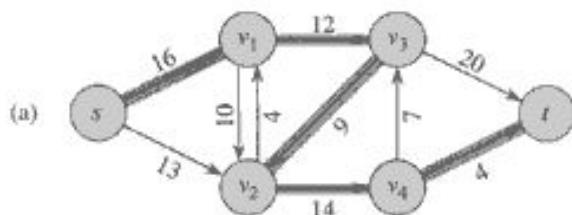
Resulting Flow = 23

No augmenting path:  
Maxflow = 23



Residual Network

# Ford-Fulkerson Algorithm: Example



# Ford-Fulkerson Algorithm: Analysis

- Performance obviously
  - depends on the augmenting paths found at each iteration
- If edge capacities are integers (or, rational numbers [*apply an appropriate scaling transformation to make them all integral*]):
  - Then the algorithm returns max-flow
  - The algorithm runs in polynomial time
- If edge capacities are irrational numbers:
  - Then the algorithm might not even terminate
  - It need not even converge to the maximum value

# Capacities

FORD-FULKERSON( $G, s, t$ )

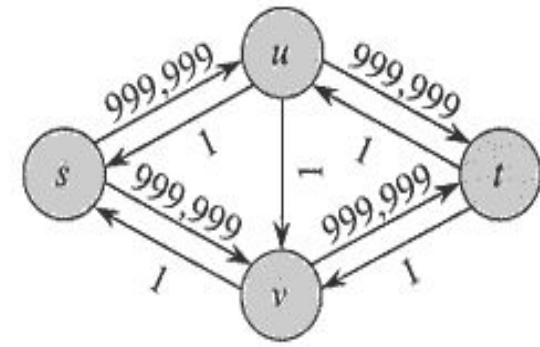
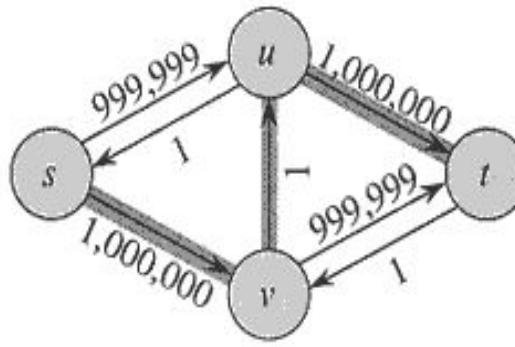
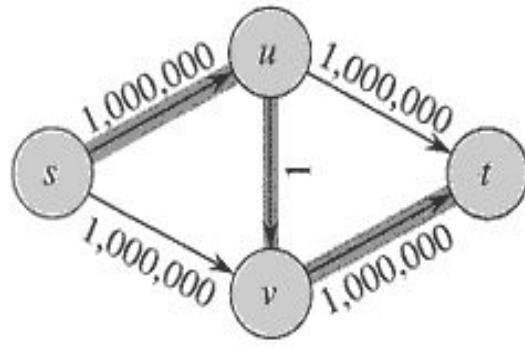
```
1  for each edge  $(u, v) \in E[G]$ 
2      do  $f[u, v] \leftarrow 0$ 
3           $f[v, u] \leftarrow 0$ 
4  while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$ 
5      do  $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v) \text{ is in } p\}$ 
6          for each edge  $(u, v)$  in  $p$ 
7              do  $f[u, v] \leftarrow f[u, v] + c_f(p)$ 
8                   $f[v, u] \leftarrow -f[u, v]$ 
```

If each  $c(e)$  is an *integer*, then time complexity is  $O(E | f^*|)$ , where  $f^*$  is the maximum flow found by the algorithm

- Lines 1-3 take  $O(E)$  time.
- The **while loop** of Lines 4-8 is executed at most  $|f^*|$  times, since the value of the flow increases by at least 1 at each iteration.
  - Each iteration of the **while loop** takes  $O(E)$  time if either depth-first or breadth-first search is used to find a path in the residual network.
- Therefore, total time =  $O(E + E | f^*|) = O(E | f^*|)$ .

# Ford-Fulkerson Algorithm: Integral Capacities

- Ford-Fulkerson algorithm runs in  $O(E |f^*|)$  time, where  $f^*$  is the maximum flow found by the algorithm
- Not really polynomial in  $|V|$  and  $|E|$ 
  - Depends on  $|f^*|$



# Edmonds-Karp Algorithm

- A small fix to the Ford-Fulkerson algorithm makes it work in polynomial time.
- Select the augmenting path using **breadth-first search** on residual network.
- The augmenting path  $p$  is the shortest path from  $s$  to  $t$  in the residual network (treating all edge weights as 1).
- Runs in time  $O(V E^2)$ .

FORD-FULKERSON( $G, s, t$ )

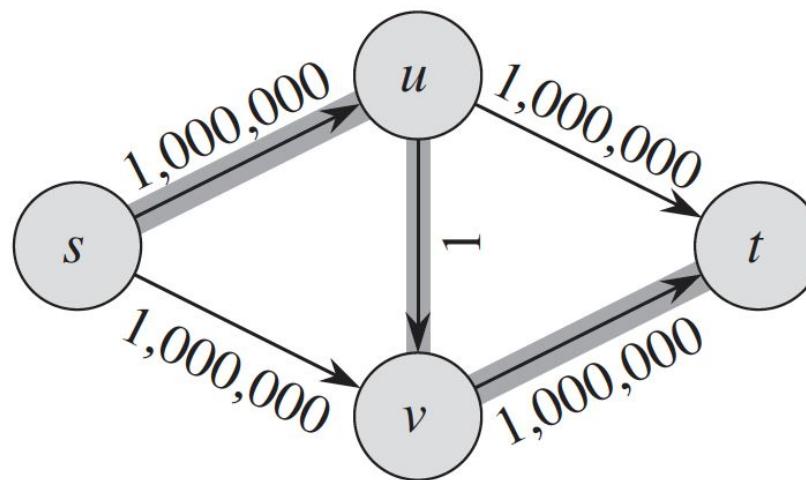
```
1  for each edge  $(u, v) \in E[G]$ 
2      do  $f[u, v] \leftarrow 0$ 
3           $f[v, u] \leftarrow 0$ 
4  while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$ 
5      do  $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v) \text{ is in } p\}$ 
6          for each edge  $(u, v)$  in  $p$ 
7              do  $f[u, v] \leftarrow f[u, v] + c_f(p)$ 
8                   $f[v, u] \leftarrow -f[u, v]$ 
```

# Edmonds-Karp Algorithm

- Use the Ford-Fulkerson framework
- Implement the computation of augmenting path
  - by using breadth-first search
  - *i.e.*, a shortest-linkage path (in no. of edges) from  $s$  to  $t$  in residual network
- Enable us to bound the time complexity
  - Mainly: the number of iterations
- Time complexity of Edmonds-Karp algorithm is  $O(V E^2)$ 
  - The number of iterations is  $O(V E)$
  - Each iteration needs  $O(E)$

# Edmonds-Karp Algorithm

- Edmonds-Karp Algorithm runs **only 2** iterations on this graph



# Time Complexity

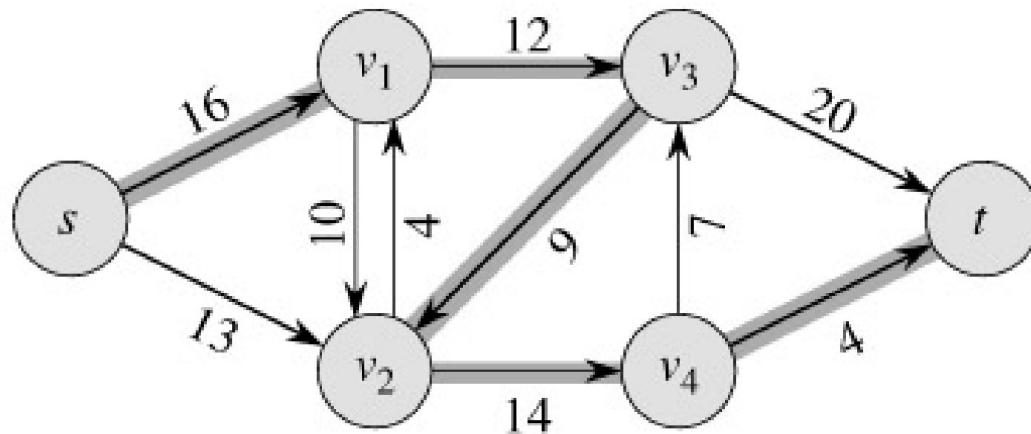
FORD-FULKERSON( $G, s, t$ )

```
1  for each edge  $(u, v) \in E[G]$ 
2      do  $f[u, v] \leftarrow 0$ 
3           $f[v, u] \leftarrow 0$ 
4  while there exists a path  $p$  from  $s$  to  $t$  in the residual network  $G_f$ 
5      do  $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v) \text{ is in } p\}$ 
6          for each edge  $(u, v)$  in  $p$ 
7              do  $f[u, v] \leftarrow f[u, v] + c_f(p)$ 
8                   $f[v, u] \leftarrow -f[u, v]$ 
```

- Let, total number of flow augmentations performed by Edmond-Karp algorithm is **O(VE)**
- BFS to find the augment path----**O(E)**
- Total Running time  **$O(VE^2)$** .

# Edmonds-Karp Algorithm

- Find the maximum flow of this flow network using Edmond-Karp algorithm (Don't use highlighted path here. It has been highlighted using Ford-Fulkerson Method)



It's Your Home Activity.  
Let's see who can do it!!1

# Observations

- Lemma: If the Edmonds-Karp algorithm is run on a flow network  $G = (V, E)$  with source  $s$  and sink  $t$ , then for all vertices  $v \in V - \{s, t\}$ , the shortest distance  $\delta_f(s, v)$  in the residual network  $G_f$  increases monotonically with each flow augmentation.

Proof:

- Theorem: If the Edmonds-Karp algorithm is run on a flow network  $G = (V, E)$  with source  $s$  and sink  $t$ , then the total number of flow augmentations performed by the algorithm is  $O(V E)$ .

Proof:

- Time complexity of Edmonds-Karp algorithm is  $O(V E^2)$

# Cuts of Flow Networks

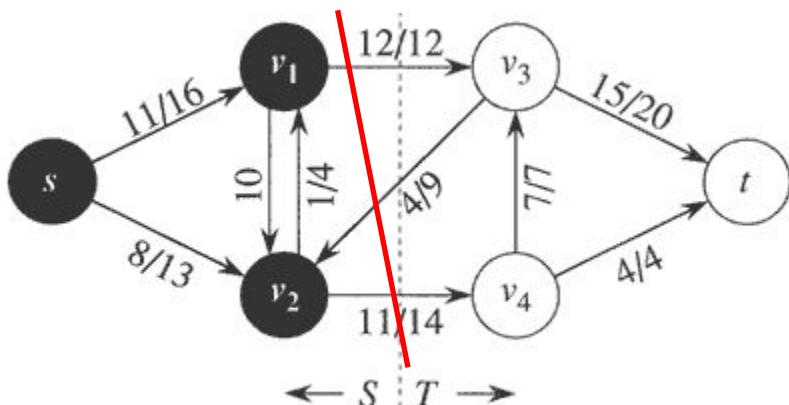
- A *cut*  $(S, T)$  of flow network  $G = (V, E)$ 
  - is a partition of  $V$  into  $S$  and  $T = V - S$ , s.t.  $s \in S$  and  $t \in T$
  - The net flow  $f(S, T)$  across cut  $(S, T)$  is

$$f(S, T) = \sum_{u \in S, v \in T} f(u, v) - \sum_{u \in S, v \in T} f(v, u)$$

- The capacity  $c(S, T)$  of cut  $(S, T)$  is

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

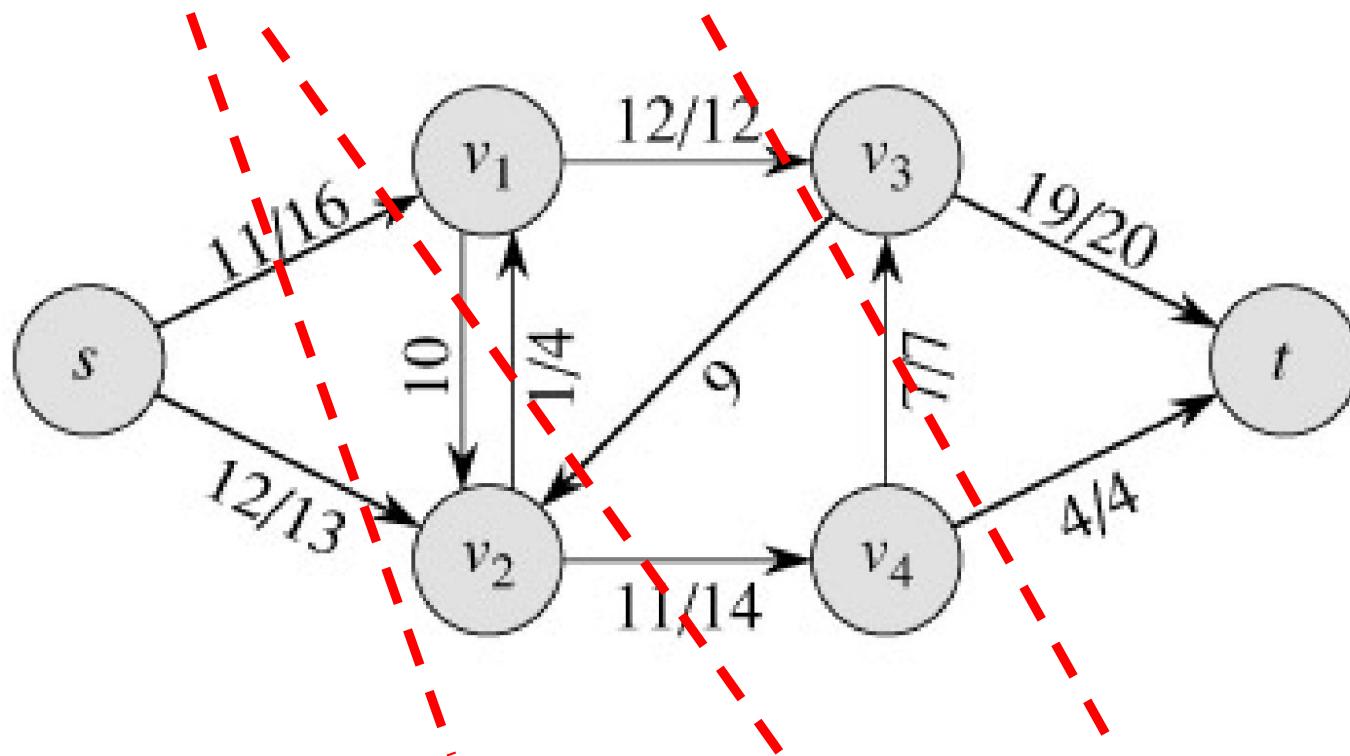
- A minimum-cut is a cut whose capacity is minimum over all cuts



$$\begin{aligned}f(S, T) &= 12 + 11 - 4 = 19 \\c(S, T) &= 12 + 14 = 26\end{aligned}$$

# Cuts of Flow Networks

- The net flow across any cut is the same and equal to the flow of the network  $|f|$ .



# Cuts of Flow Networks

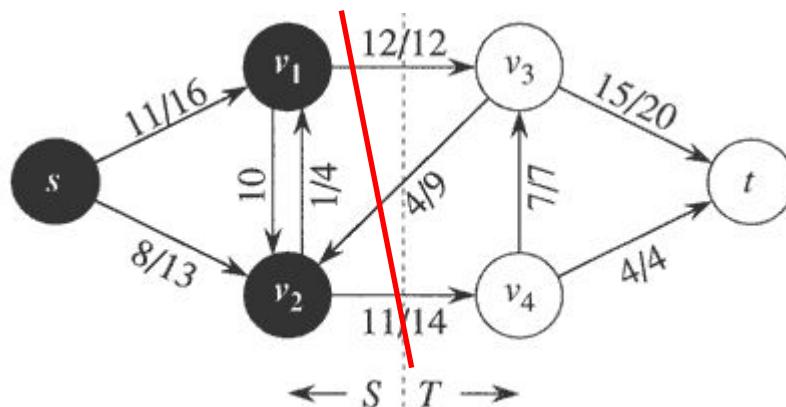
*Lemma:*

Let  $f$  be a flow in a network  $G$  with source  $s$  and sink  $t$ , and let  $(S, T)$  be a cut of  $G$ . Then the net flow across  $(S, T)$  is  $f(S, T) = |f|$ .

*Corollary:*

The value of any flow  $f$  in a flow network  $G$  is bounded from above by the capacity of any cut of  $G$ .

The value of any flow  $\leq$  the capacity of any cut



# Max-flow Min-cut Theorem

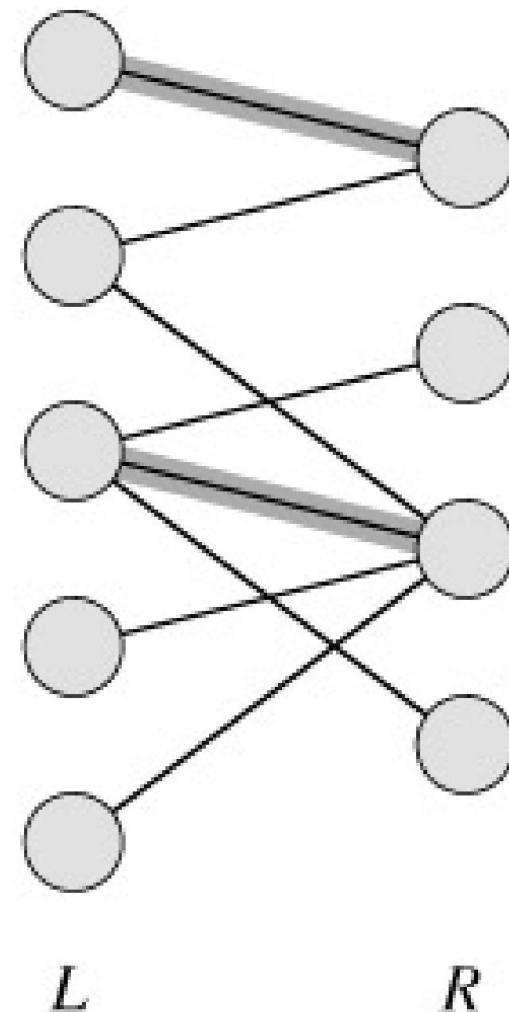
- If  $f$  is a flow in a flow network  $G = (V, E)$  with source  $s$  and sink  $t$ , then the following conditions are equivalent:
  - (1)  $f$  is a maximum flow in  $G$
  - (2) The residual network  $G_f$  contains no augmenting paths
  - (3)  $|f| = c(S, T)$  for some cut  $(S, T)$  in  $G$
- Proof:
  - . (1)  $\Rightarrow$  (2)
  - . (2)  $\Rightarrow$  (3)
  - . (3)  $\Rightarrow$  (1)

**An Application of Max Flow:**

**Maximum Bipartite Matching**

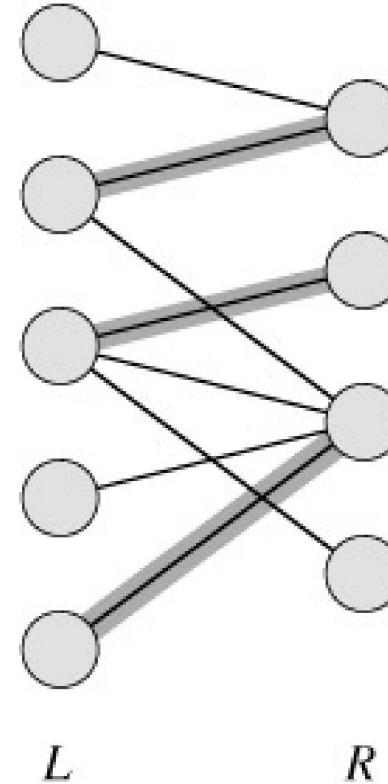
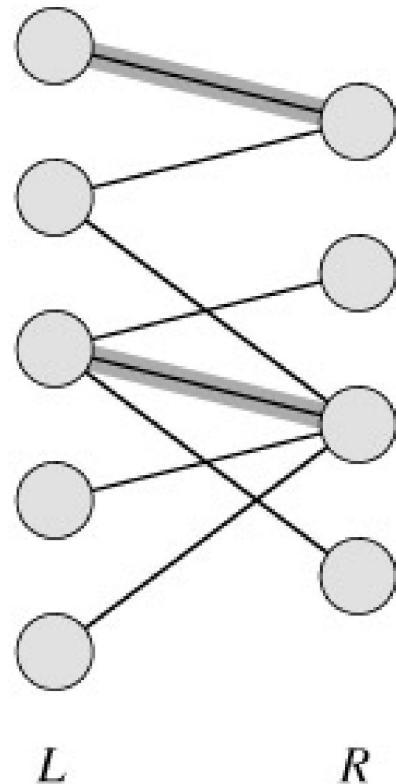
# Maximum Bipartite Matching

- A **bipartite graph** is a graph  $G = (V, E)$  in which  $V$  can be partitioned into two parts  $L$  and  $R$  such that every edge in  $E$  is between a vertex in  $L$  and a vertex in  $R$ .
- e.g. vertices in  $L$  represent skilled workers and vertices in  $R$  represent jobs. An edge connects workers to jobs they can perform.



# Matching

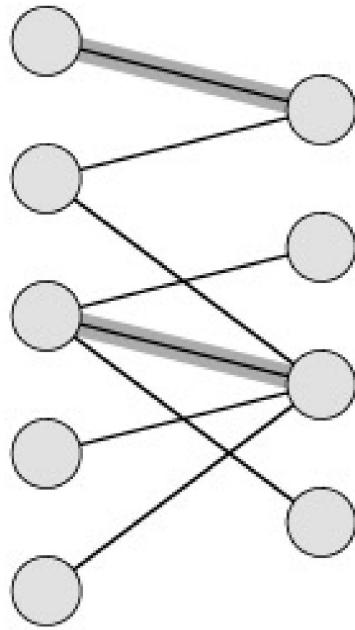
- A **matching** in a graph is a subset  $M$  of  $E$ , such that for all vertices  $v$  in  $V$ , at most one edge of  $M$  is incident on  $v$ .



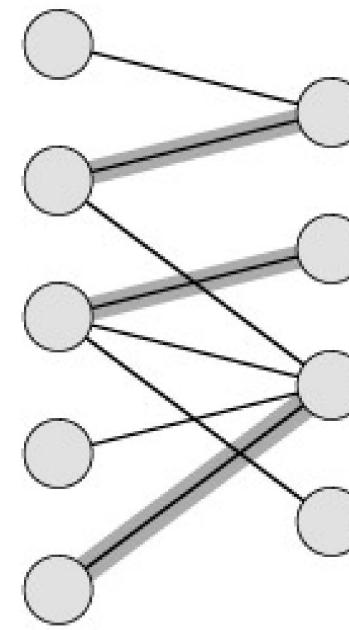
# Maximum Matching

- A **maximum matching** is a matching of maximum cardinality (maximum number of edges).

not maximum

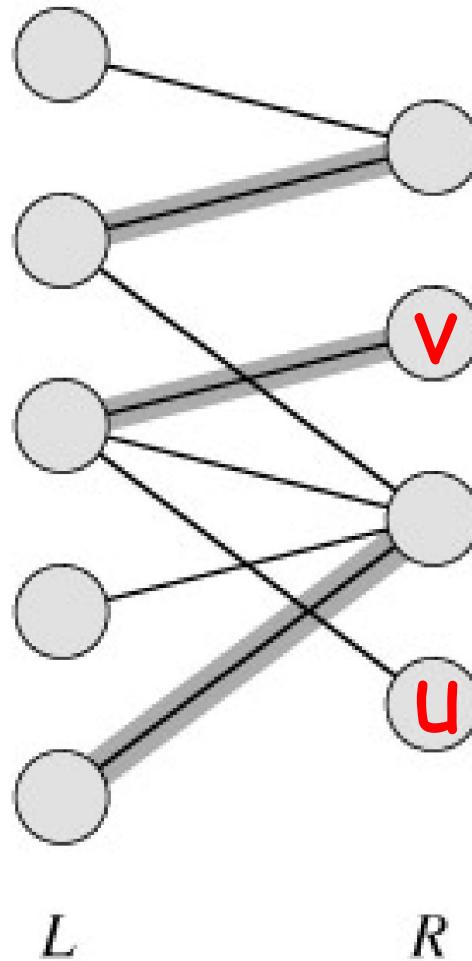


maximum



# Maximum Matching

- No matching of cardinality 4, because only one of v and u can be matched.
- In the workers-jobs example a max-matching provides work for as many people as possible.

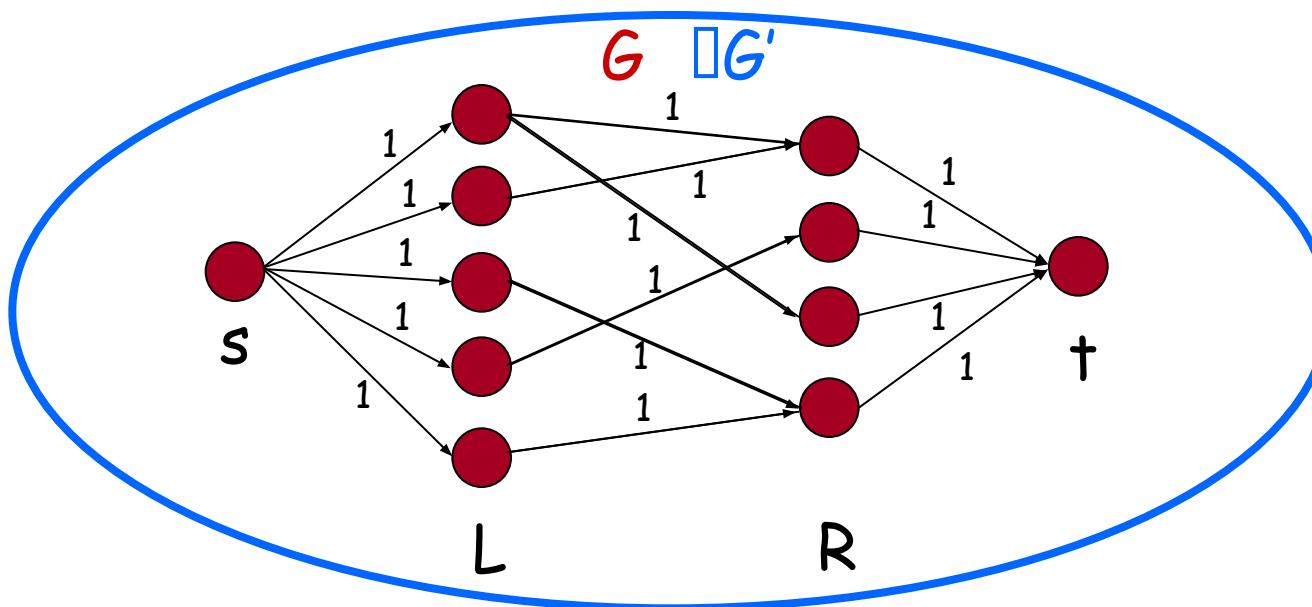


## Solving the Maximum Bipartite Matching Problem

- Reduce the maximum bipartite matching problem on graph **G** to the max-flow problem on a corresponding flow network **G'**.
- Solve using Ford-Fulkerson algorithm.

# Corresponding Flow Network

- To form the corresponding flow network  $\mathbf{G}'$  of the bipartite graph  $\mathbf{G}$ :
  - Add a source vertex  $s$  and edges from  $s$  to  $L$ .
  - Direct the edges in  $E$  from  $L$  to  $R$ .
  - Add a sink vertex  $t$  and edges from  $R$  to  $t$ .
  - Assign a capacity of 1 to all edges.
- Claim: max-flow in  $\mathbf{G}'$  corresponds to a max-bipartite-matching on  $\mathbf{G}$ .



# Solving Bipartite Matching as Max Flow

Let  $G = (V, E)$  be a bipartite graph with vertex partition  $V = L \cup R$ .

Let  $G' = (V', E')$  be its corresponding flow network.

If  $M$  is a matching in  $G$ ,

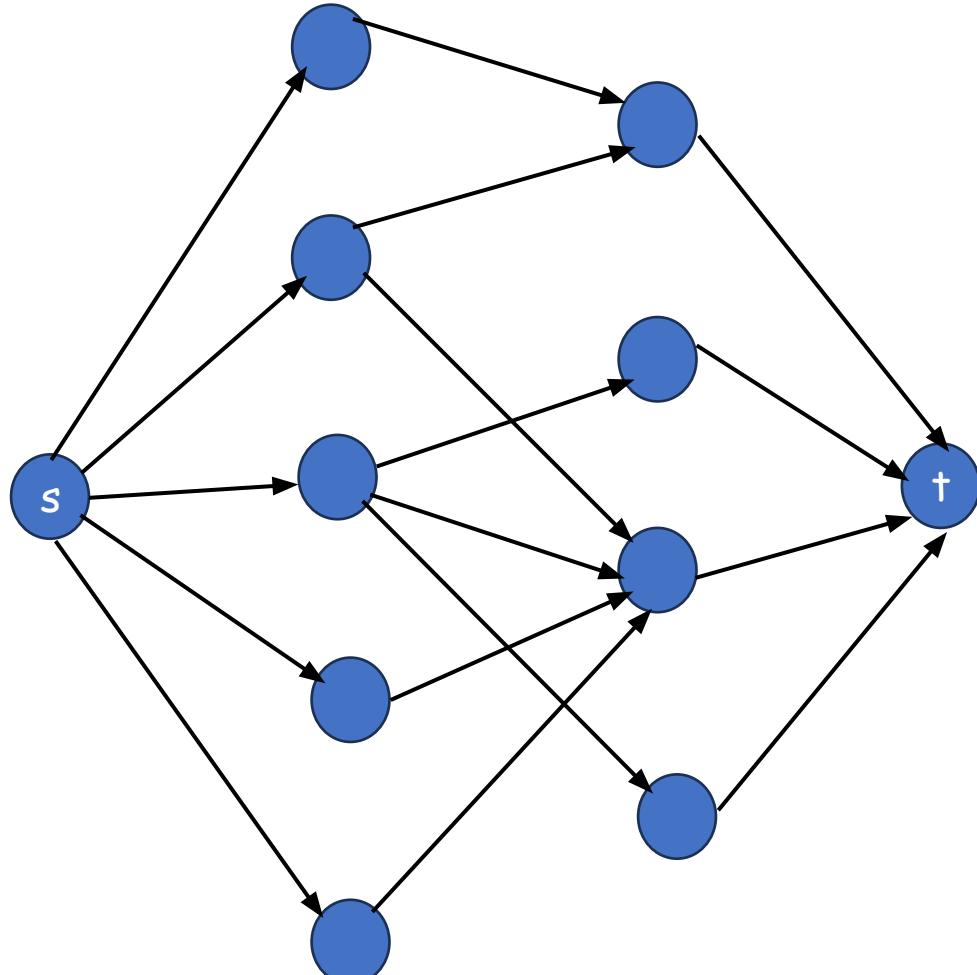
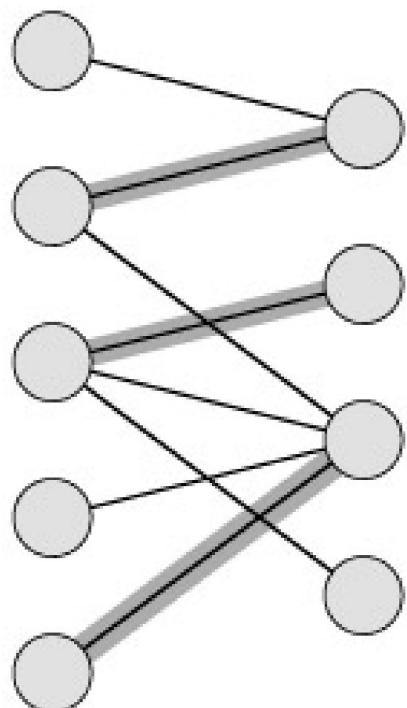
then there is an integer-valued flow  $f$  in  $G'$  with value  $|f| = |M|$ .

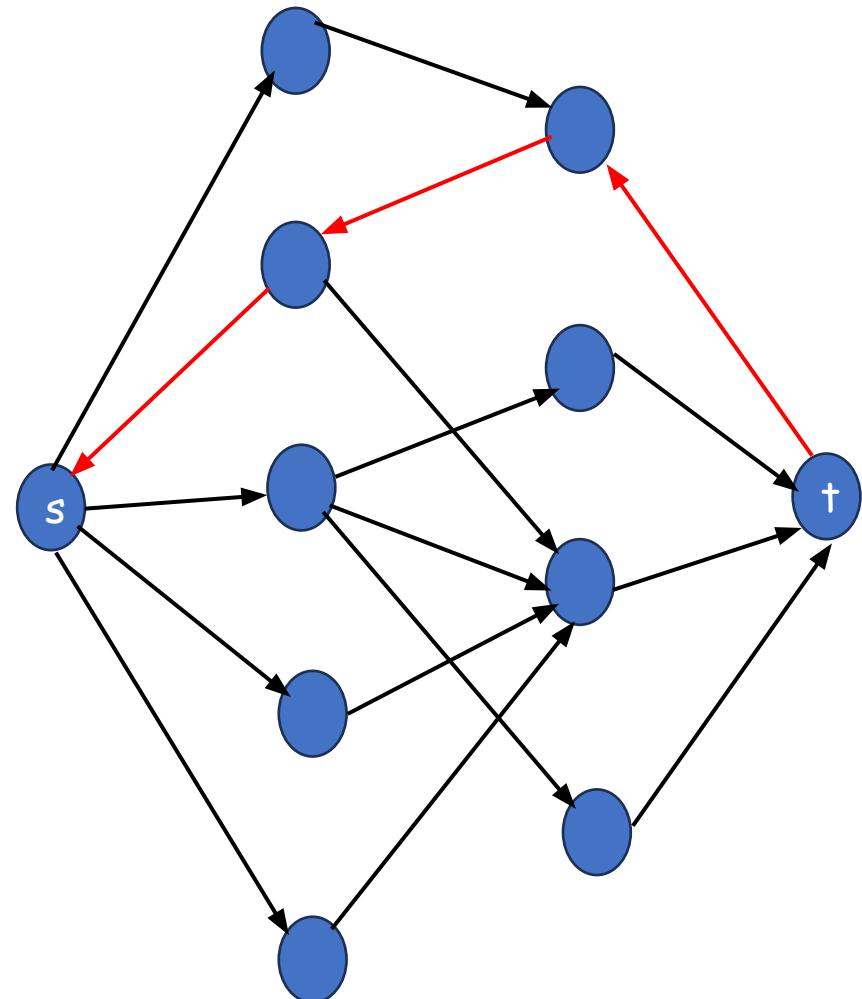
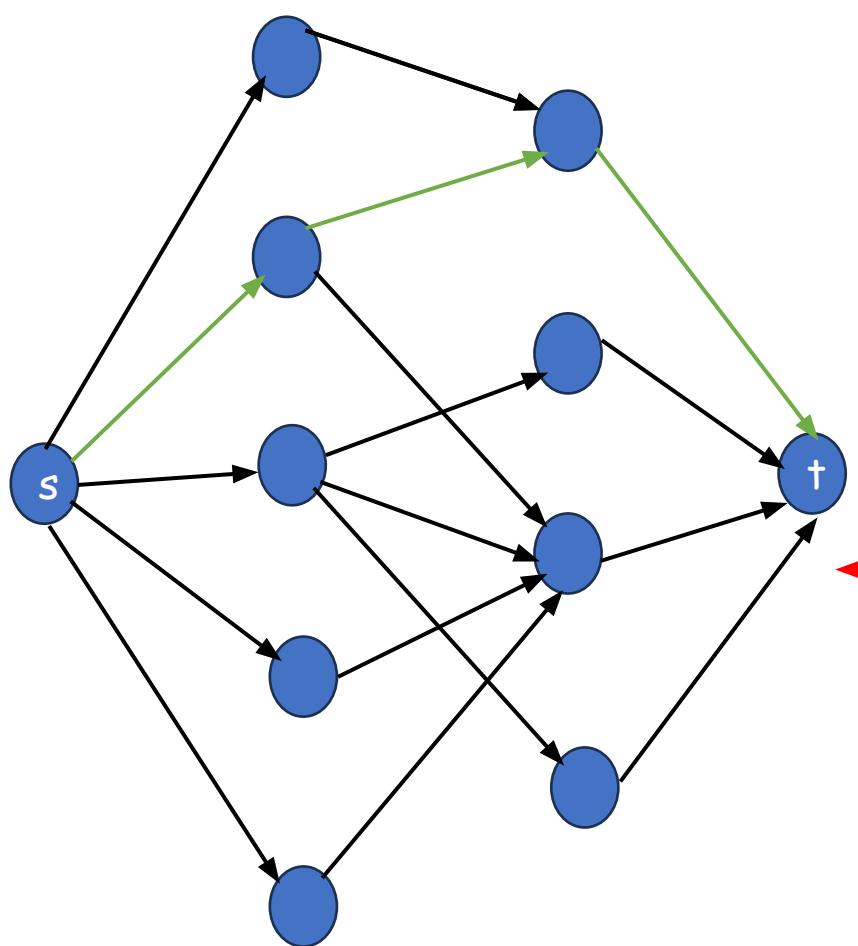
Conversely, if  $f$  is an integer-valued flow in  $G'$ ,

then there is a matching  $M$  in  $G$  with cardinality  $|M| = |f|$ .

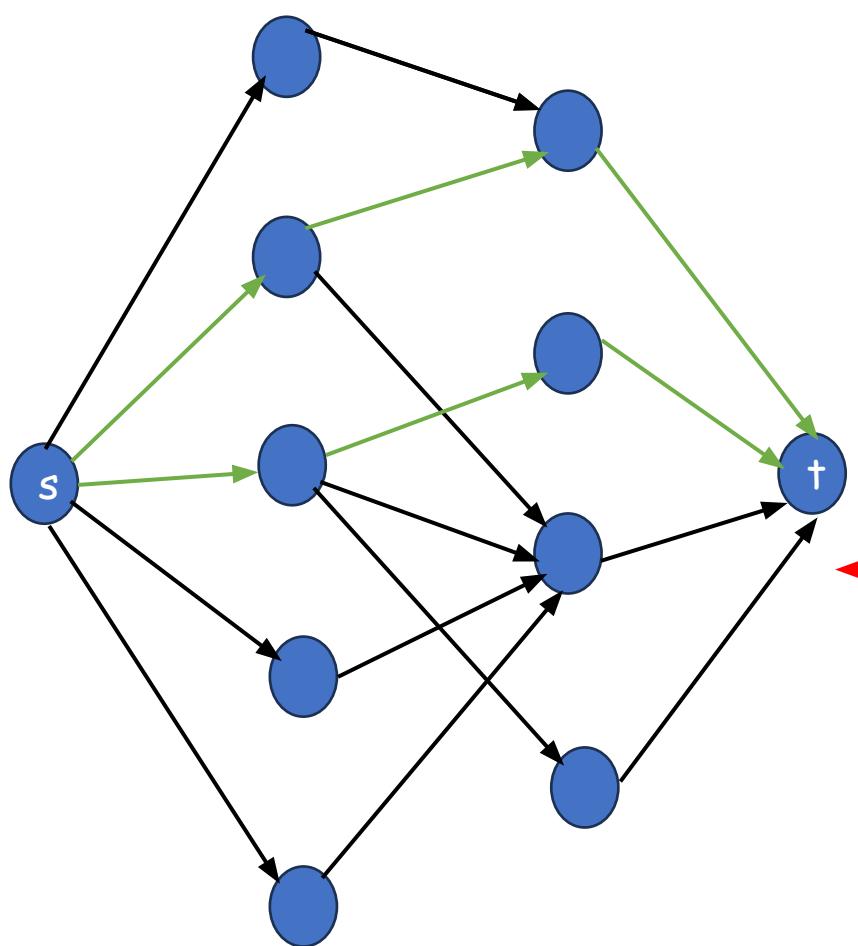
Thus  $\max |M| = \max(\text{integer } |f|)$

# Example

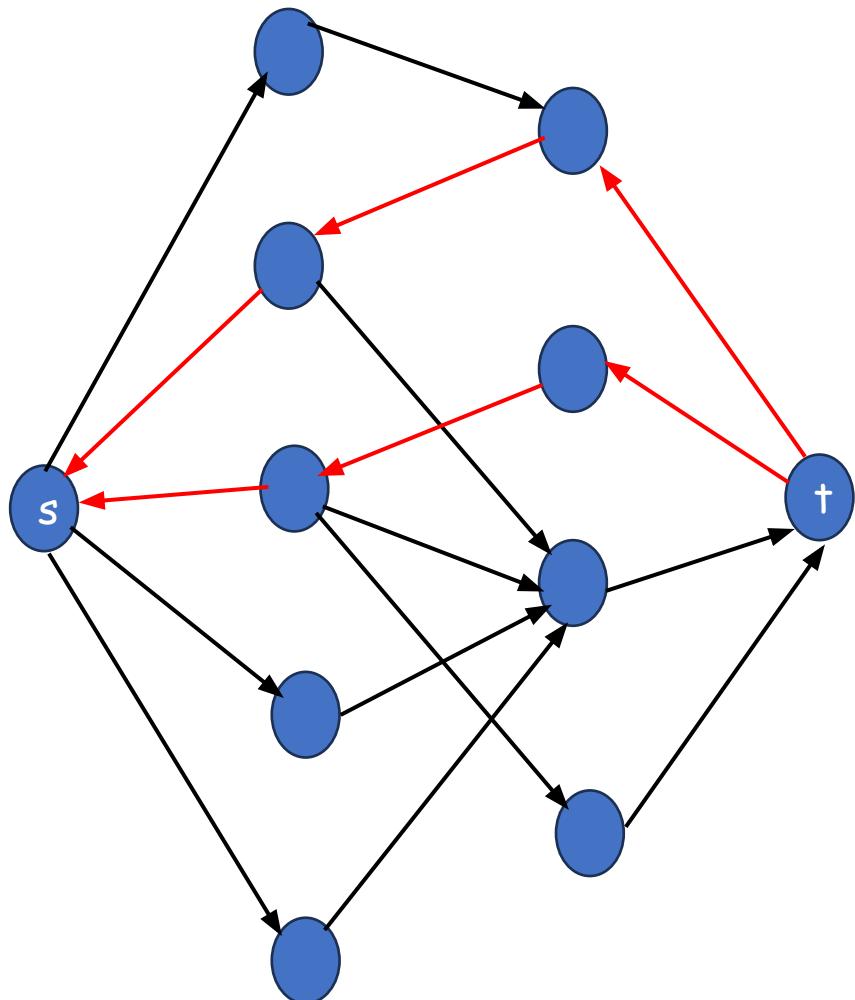


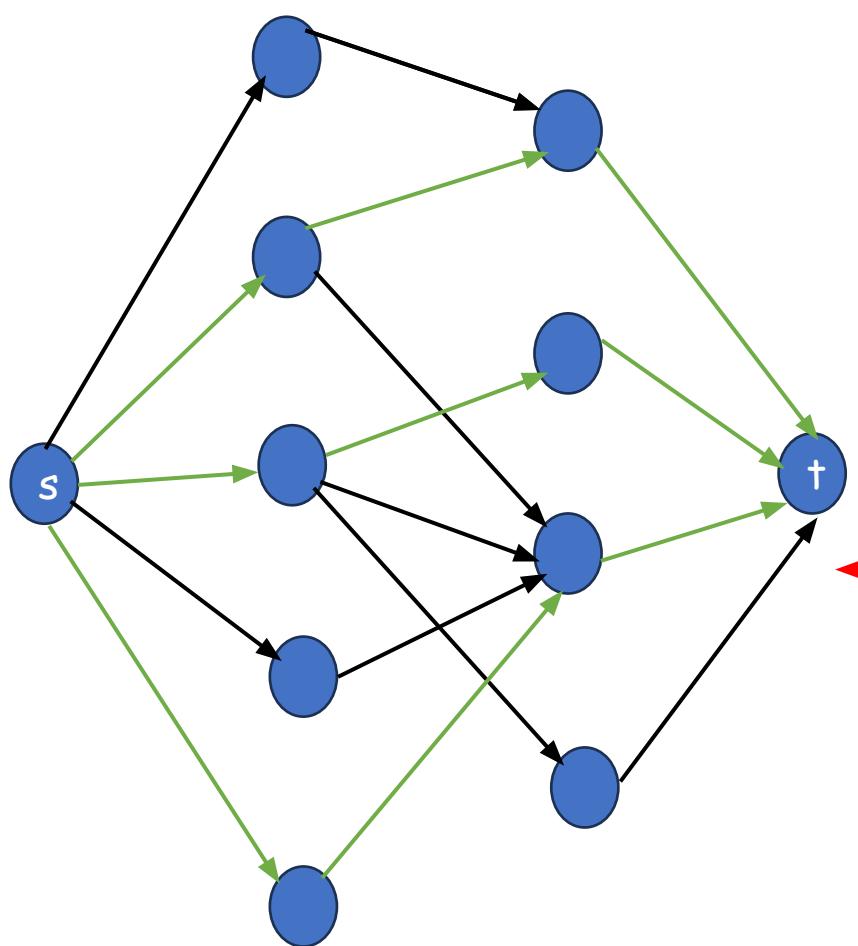


Residual  
Network

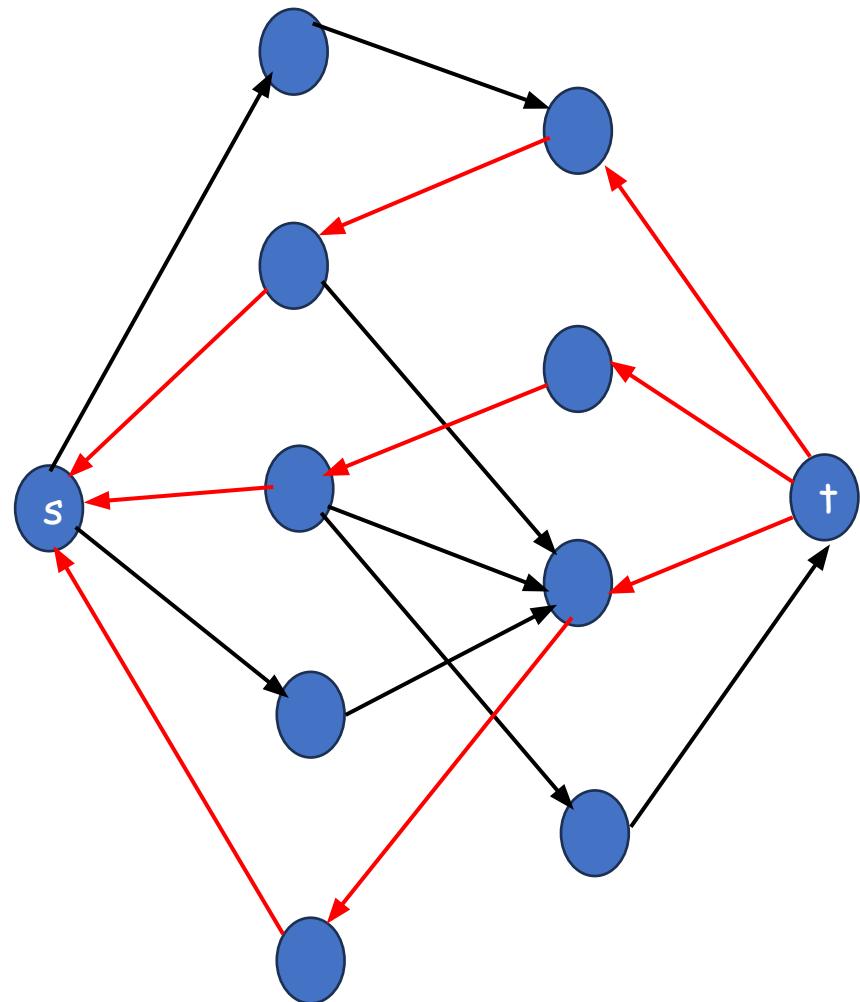


Residual  
Network

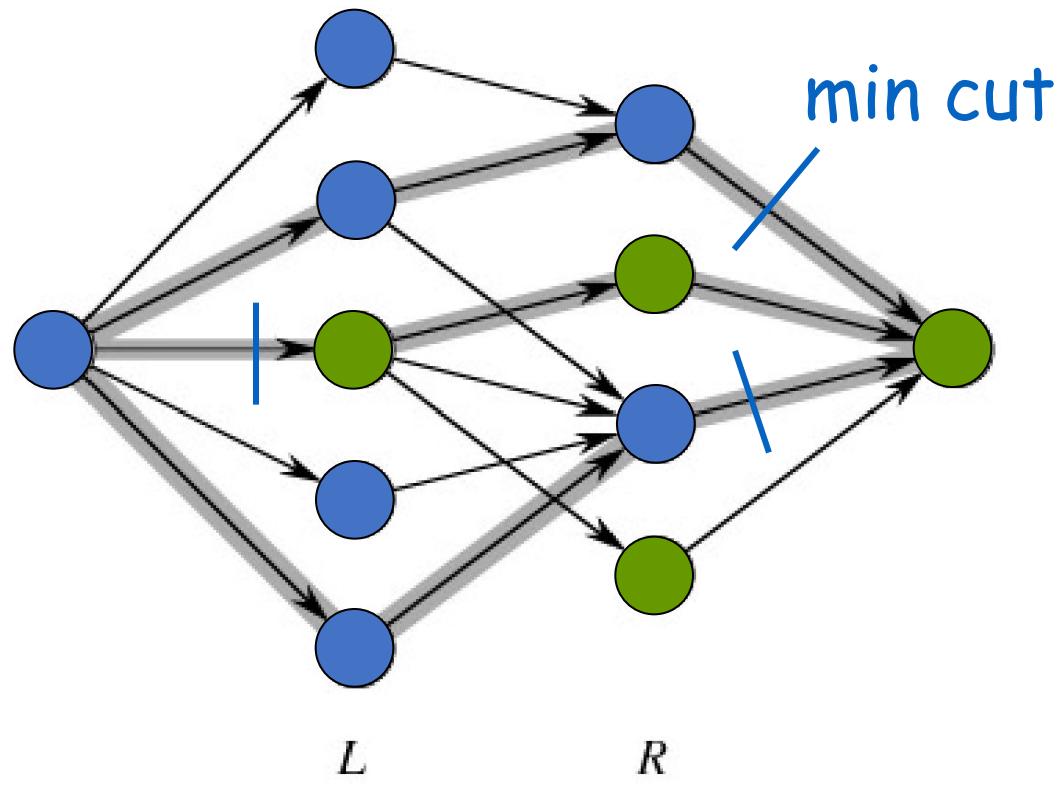
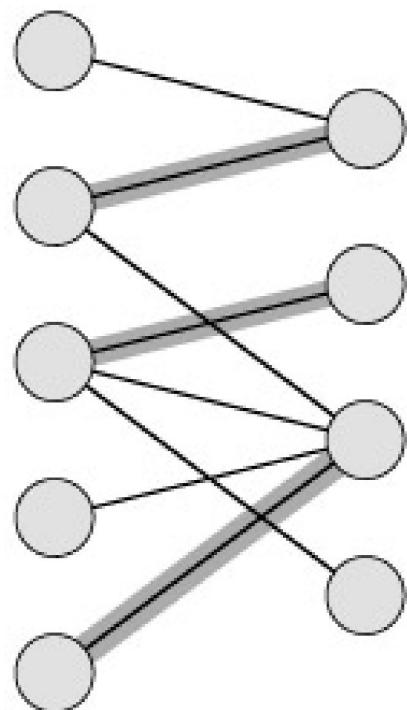




Residual  
Network



# Example



$L$

$L$

$R$

$$|M| = 3$$

$$\square\square$$

$$\text{max flow} = |f| = 3$$

# Conclusion

- Network flow algorithms allow us to find the maximum bipartite matching fairly easily.
- Similar techniques are applicable in other combinatorial design problems.

# Useful Links

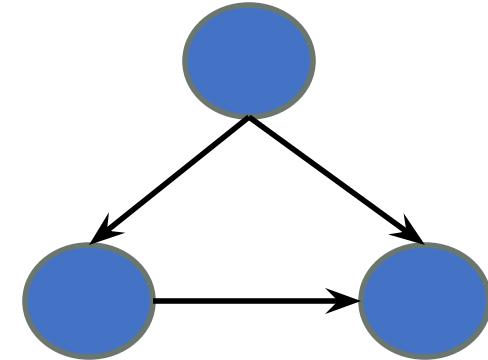
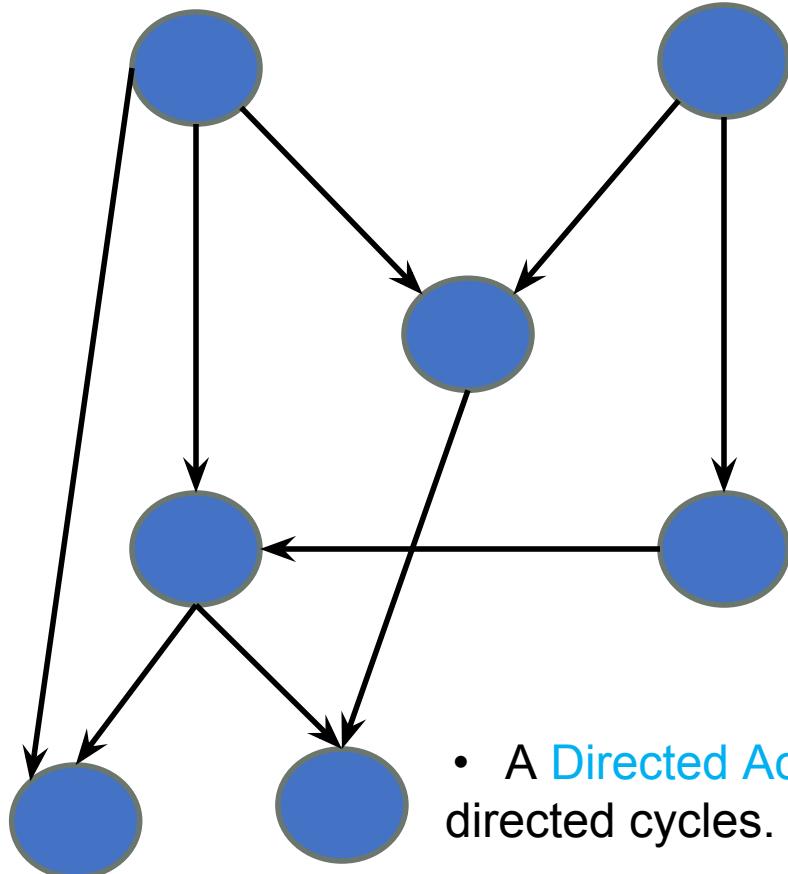
- <https://www.youtube.com/watch?v=70N1N8gPtU>
- <https://www.youtube.com/watch?v=NwenwITjMys&t=825s>
- <https://www.youtube.com/watch?v=HdJQ0qpbIDs>

# CSE 215: Data Structures and Algorithms II

---

Topological Sorting  
Strongly Connected Components<sub>1</sub>

# Directed Acyclic Graph



- A **Directed Acyclic Graph** or **DAG** is a directed graph with no directed cycles.

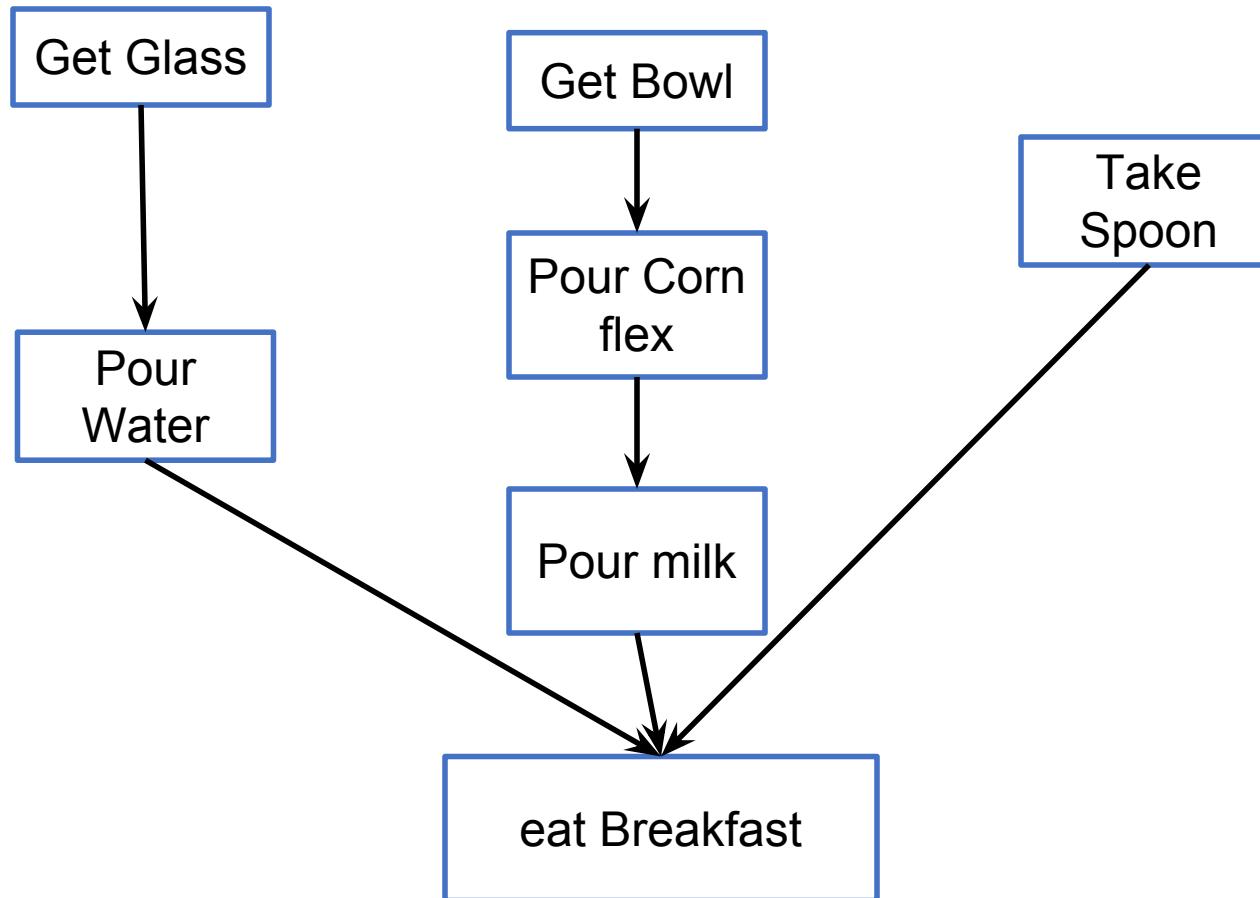
# Topological Sort

- A *topological sort* of a **DAG** is a linear ordering of all vertices of the graph  $G$  such that vertex  $u$  comes before vertex  $v$  if  $(u, v)$  is an edge in  $G$ .
- DAG indicates precedence among events:  
events are graph vertices, edge from  $u$  to  $v$  means event  $u$  has precedence over event  $v$
- Real-world example:
  - getting dressed
  - course registration
  - tasks for eating meal

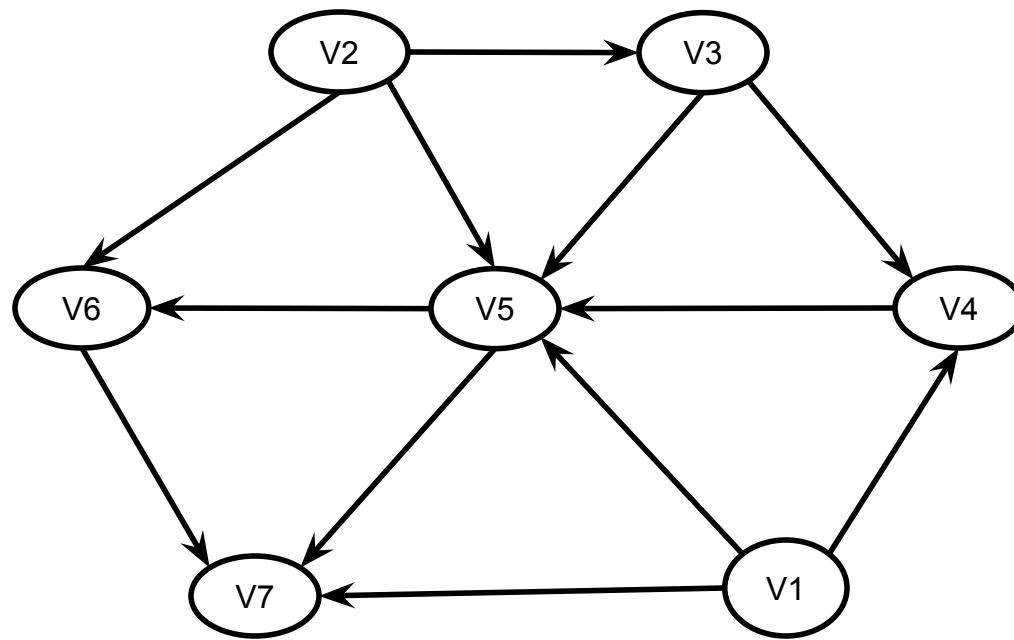
# Precedence Example

- Tasks that have to be done to eat breakfast:
  - get glass, pour juice, get bowl, pour cereal, pour milk, get spoon, eat.
- Certain events must happen in a certain order (ex: get bowl before pouring milk)
- For other events, it doesn't matter (ex: get bowl and get spoon)

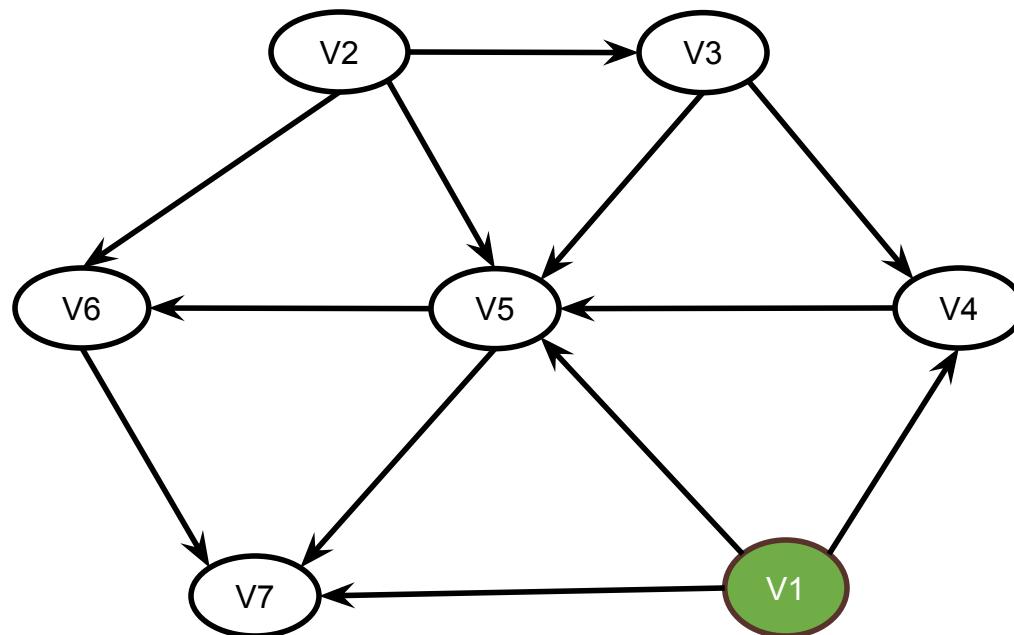
# Precedence Example



# Topological Sort: Using in-degree

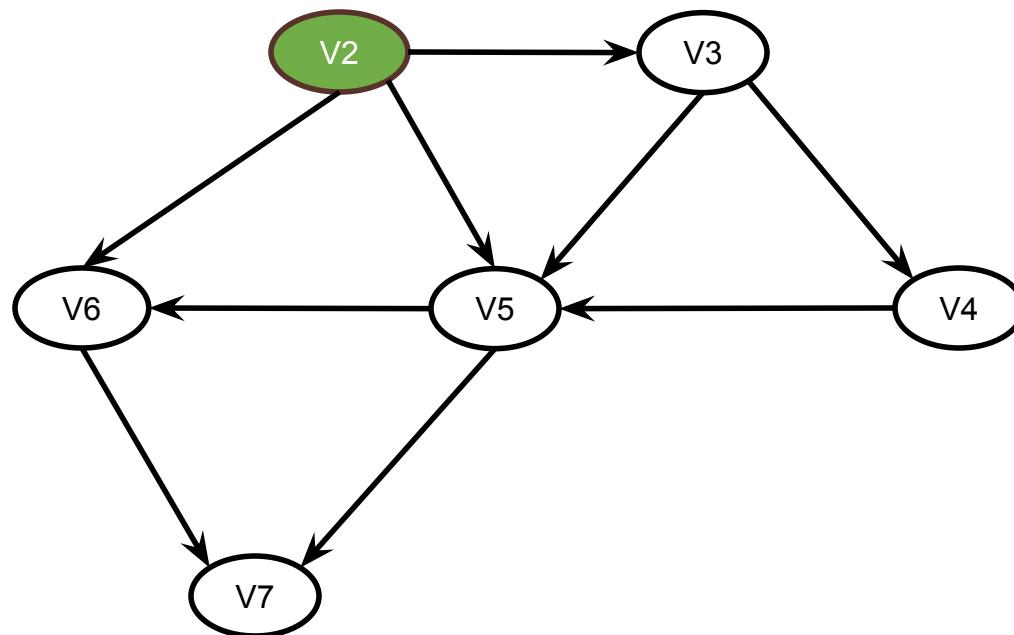


# Topological Sort: Using in-degree



Delete the vertex whose in-degree 0. (v1 or v2)

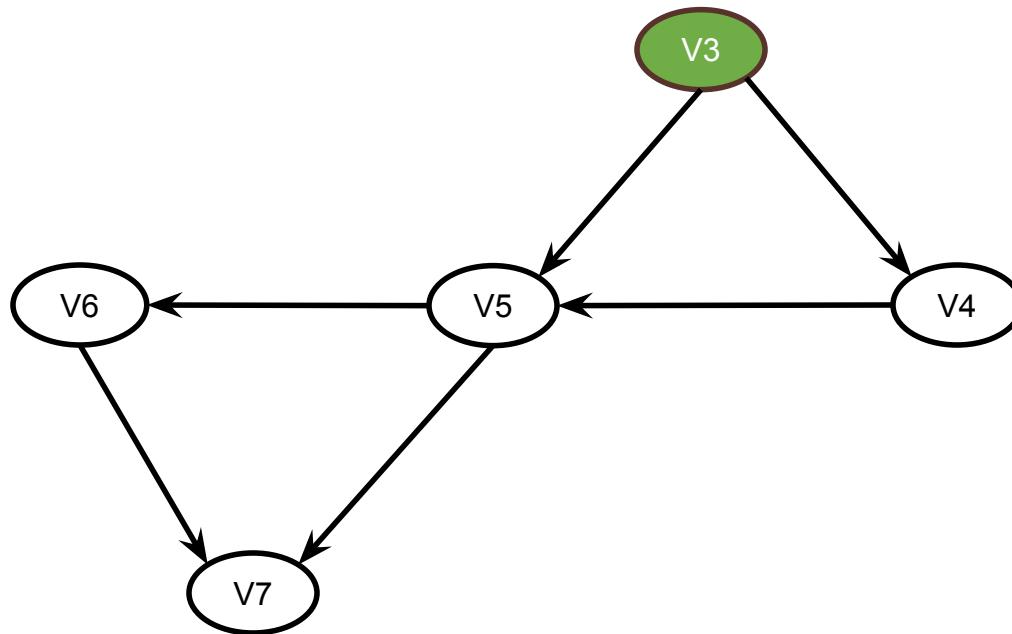
# Topological Sort: Using in-degree



T: v1

Delete the vertex whose in-degree 0. (v2)

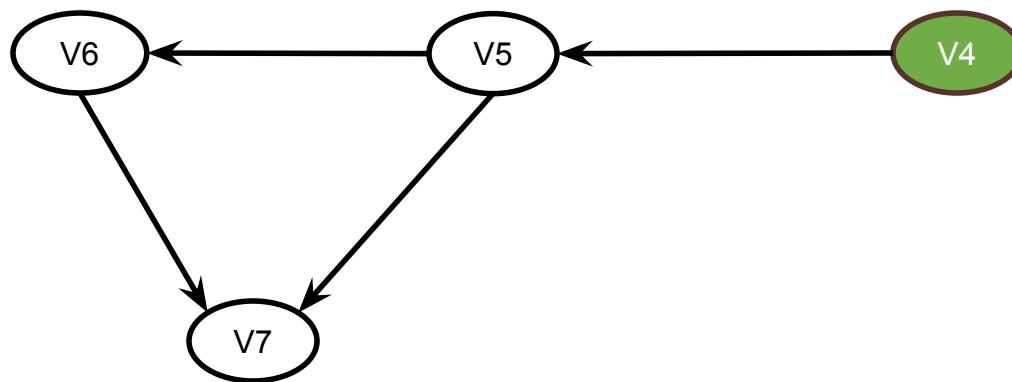
# Topological Sort: Using in-degree



T: V1 V2

Delete the vertex whose in-degree 0. (v3)

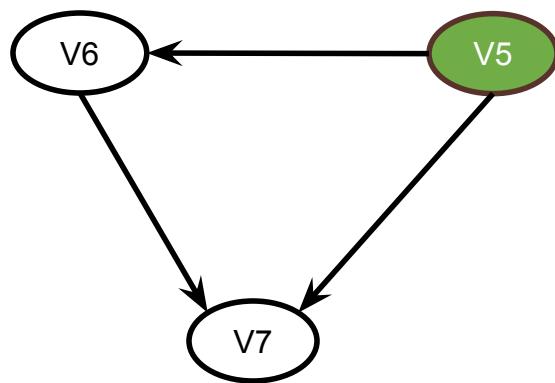
# Topological Sort: Using in-degree



T: v1 v2 v3

Delete the vertex whose in-degree 0. (v4)

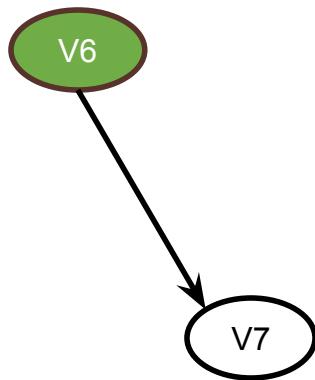
# Topological Sort: Using in-degree



T: v1 v2 v3 v4

Delete the vertex whose in-degree 0. (v5)

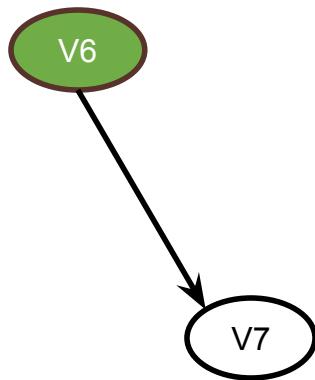
# Topological Sort: Using in-degree



T: v1 v2 v3 v4 v5

Delete the vertex whose in-degree 0. (v6)

# Topological Sort: Using in-degree



T: v1 v2 v3 v4 v5 v6

Delete the vertex whose in-degree 0. (v6)

# Topological Sort: Using in-degree

v7

T: v1 v2 v3 v4 v5 v6

Delete the vertex whose in-degree 0. (v7)

# Topological Sort: Using in-degree

T: v1 v2 v3 v4 v5 v6 v7

Delete the vertex whose in-degree 0.

# Topological Sort: Using in-degree

Steps for finding the topological ordering of a DAG:

**Step-1:** Compute **in-degree** for each of the vertices present in the DAG and initialize the count of visited nodes as 0;

**Step-2:** Add all **vertices with in-degree equals 0** into a queue

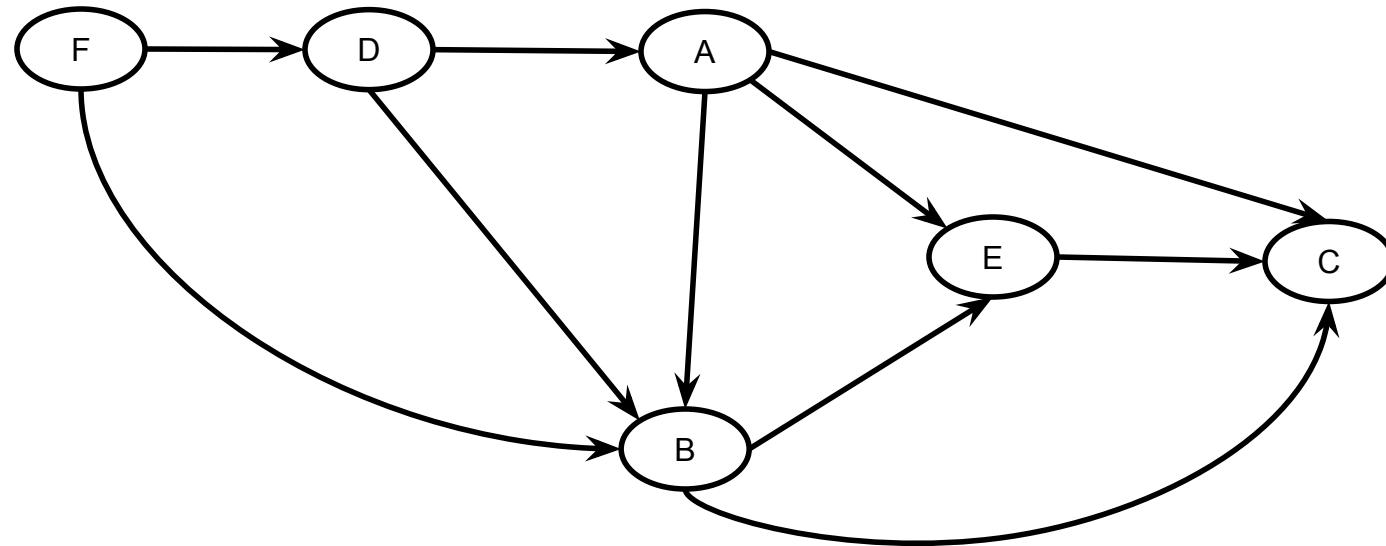
**Step-3:** Remove a vertex from the queue and then

- Increment count of visited nodes by 1;
- Decrease in-degree by 1 for all its neighboring nodes;
- If in-degree of a neighboring node is reduced to zero, then add it to the queue;

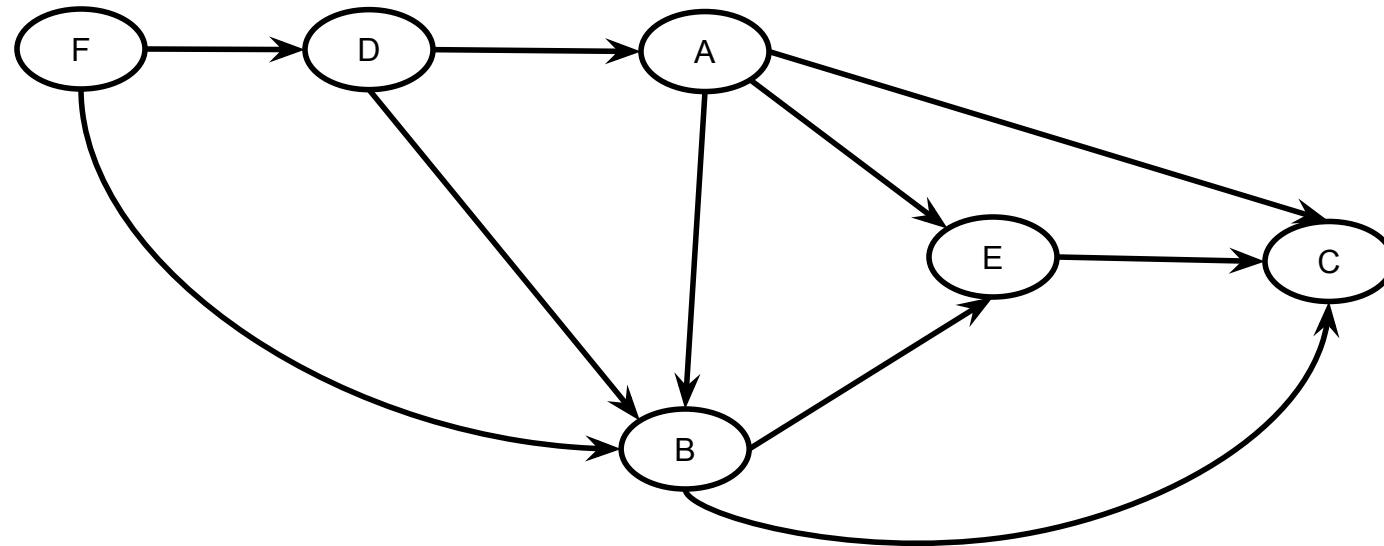
**Step 4:** Repeat Step 3 until **the queue is empty**;

**Step 5:** If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph

# Topological Sort

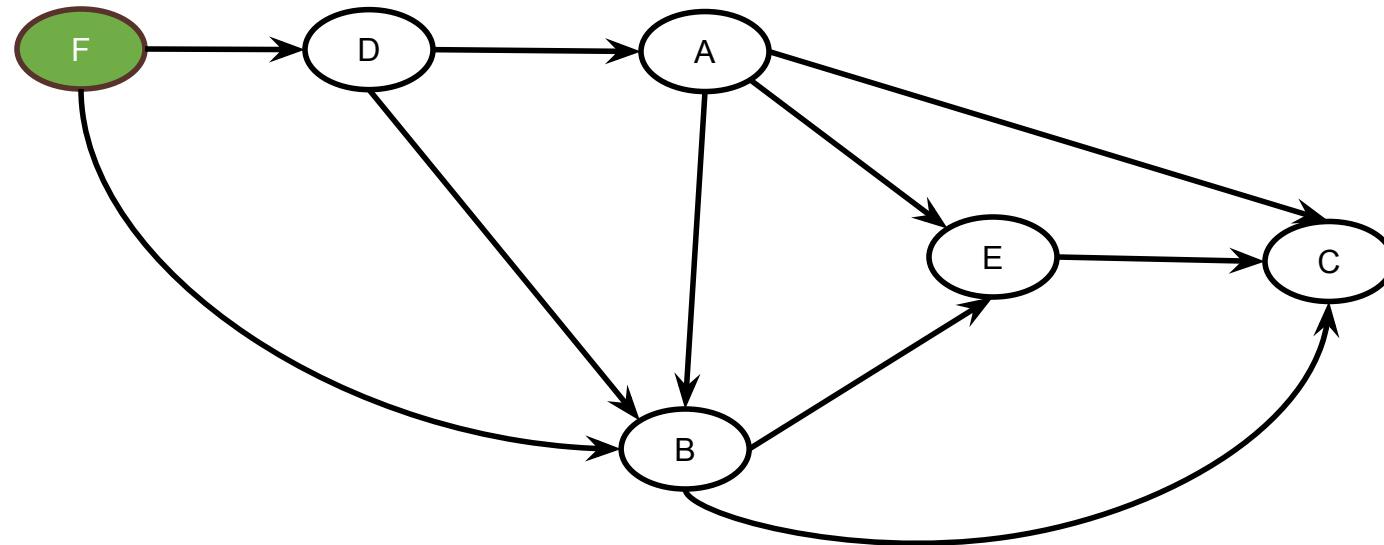


# Topological Sort



In-degree	1	3	3	1	2	0
	A	B	C	D	E	F

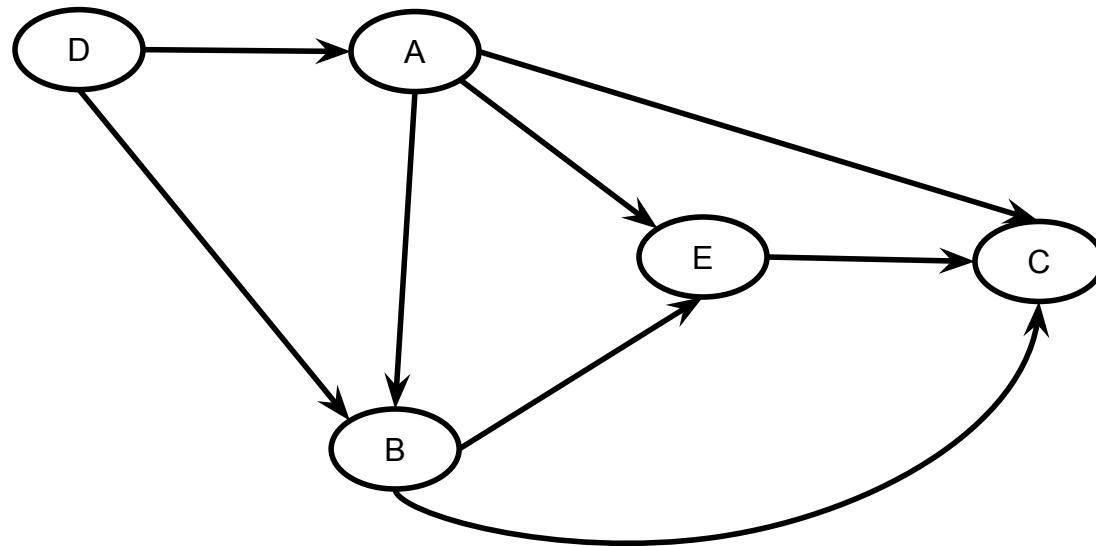
# Topological Sort



In-degree	1	3	3	1	2	0
	A	B	C	D	E	F

Q: F

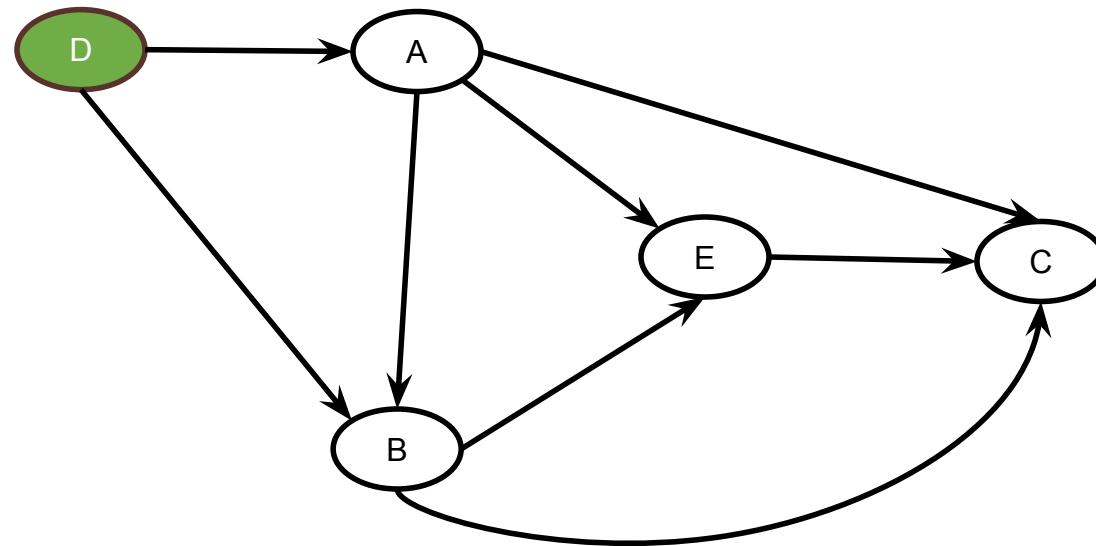
# Topological Sort



In-degree	1	2	3	0	2	0
	A	B	C	D	E	F

Q: F

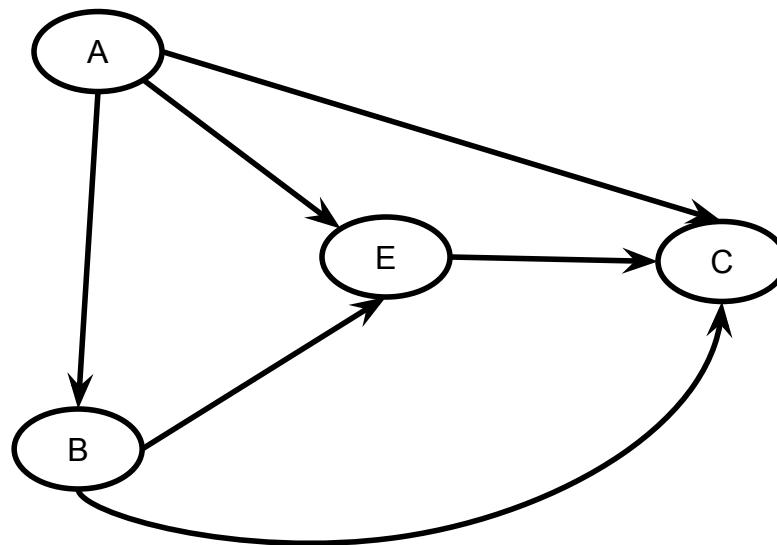
# Topological Sort



In-degree	1	2	3	0	2	0
	A	B	C	D	E	F

Q: F D

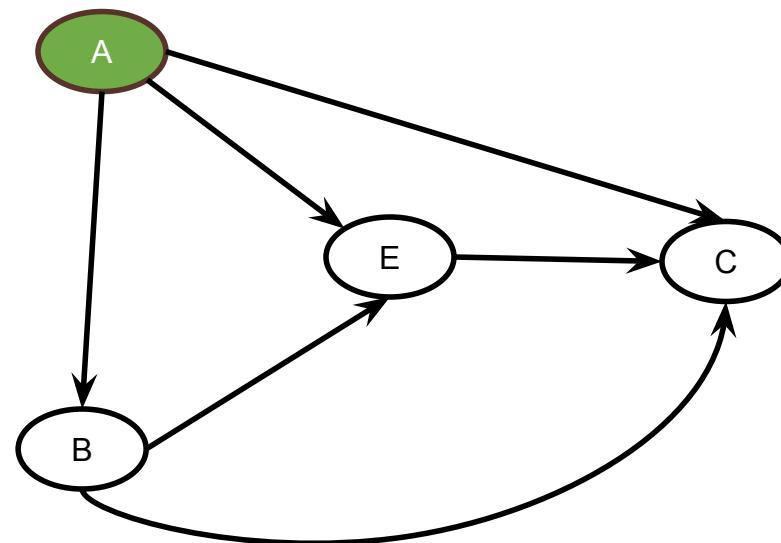
# Topological Sort



In-degree	0	1	3	0	2	0
	A	B	C	D	E	F

Q: F D

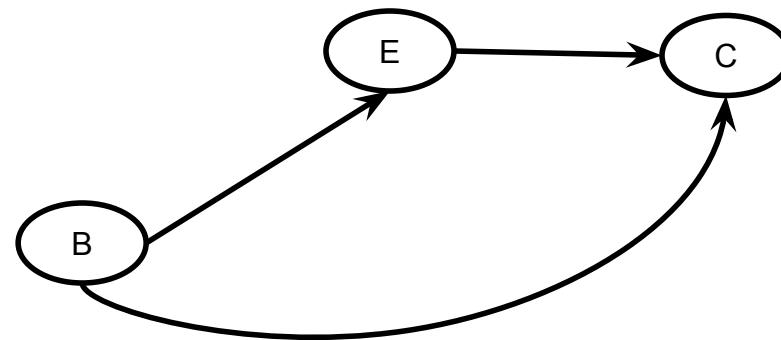
# Topological Sort



In-degree	0	1	3	0	2	0
	A	B	C	D	E	F

Q: F D A

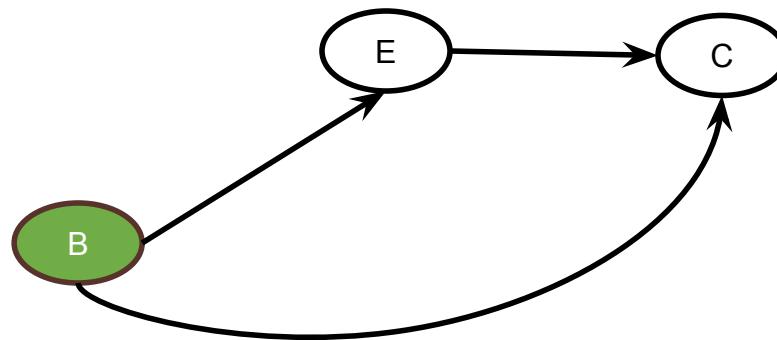
# Topological Sort



In-degree	0	0	2	0	1	0
	A	B	C	D	E	F

Q: F D A

# Topological Sort



In-degree	0	0	2	0	1	0
	A	B	C	D	E	F

Q: F D A  
B

# Topological Sort



In-degree	0	0	1	0	0	0
	A	B	C	D	E	F

Q: F D A  
B

# Topological Sort



In-degree	0	0	1	0	0	0
	A	B	C	D	E	F

Q: F D A B E

# Topological Sort



In-degree	0	0	0	0	0	0
	A	B	C	D	E	F

Q: F D A B E

# Topological Sort



In-degree	0	0	0	0	0	0
	A	B	C	D	E	F

Q: F D A B E C

# Topological Sort

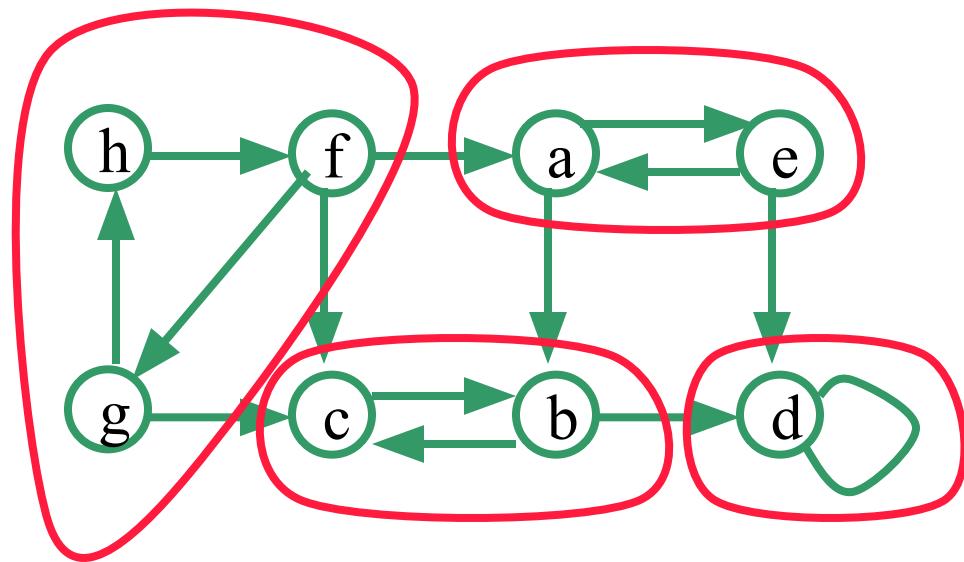
In-degree	0	0	0	0	0	0
	A	B	C	D	E	F

Q: F D A B E C

# DFS Application: Strongly Connected Components

- Consider a **directed** graph  $G$ .
- A **strongly connected component (SCC)** of the graph  $G$  is a maximal set of vertices with a (directed) path between every pair of vertices
- Problem: Find all the SCCs of the graph.

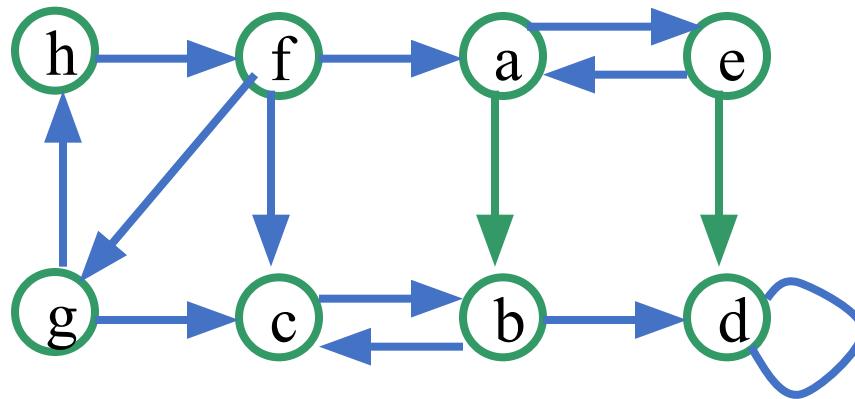
# SCC Example



four  
SCCs

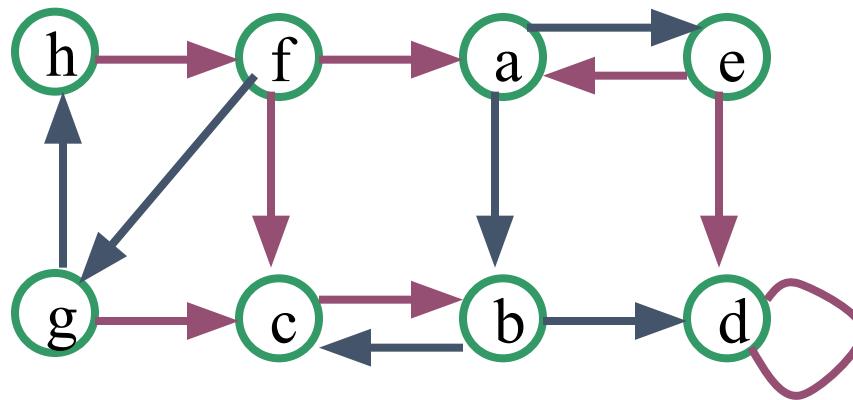
# How Can DFS Help?

- Suppose we run DFS on the directed graph.
- All vertices in the same SCC are in the same DFS tree.
- But there might be several different SCCs in the same DFS tree.
  - Example: start DFS from the vertex  $a$  in the following graph



# How Can DFS Help?

- Suppose we run DFS on the directed graph.
- All vertices in the same SCC are in the same DFS tree.
- But there might be several different SCCs in the same DFS tree.
  - Example: start DFS from the vertex *a* in the following graph



# Main Idea of SCC Algorithm

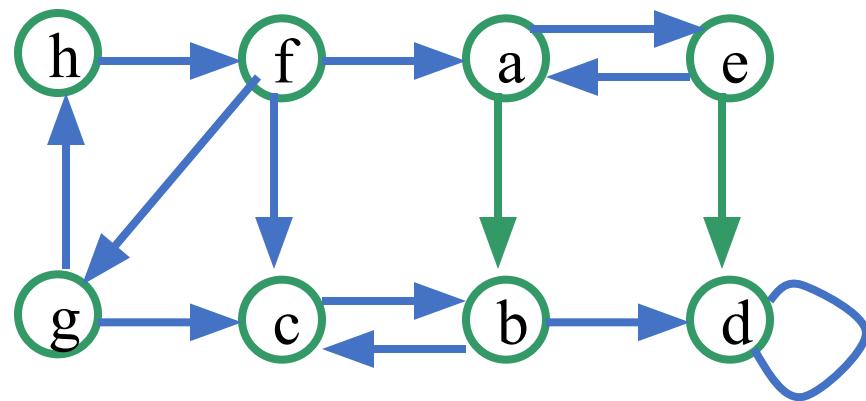
- DFS tells us which vertices are reachable from the roots of the individual trees
- Also need information in the "other direction": is the root reachable from its descendants?
- Run DFS again on the "transpose" graph (reverse the directions of the edges)

# SCC Algorithm

input: directed graph  $G = (V, E)$

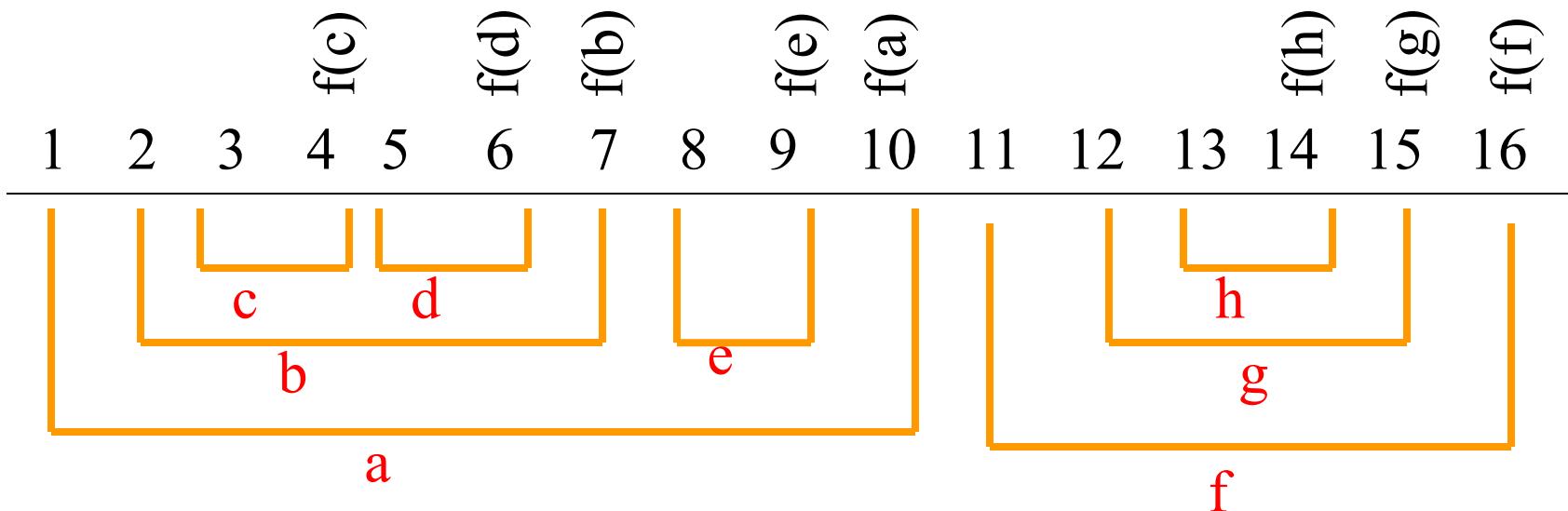
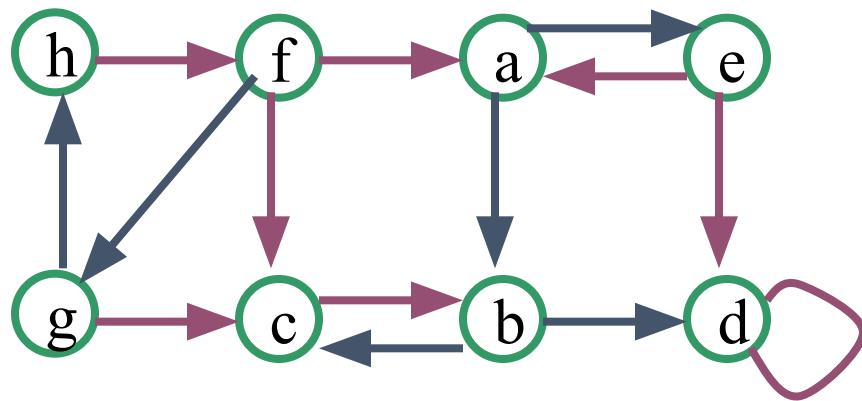
1. call  $\text{DFS}(G)$  to compute finishing times
2. compute  $G^T$  // transpose graph
3. call  $\text{DFS}(G^T)$ , considering vertices in decreasing order of finishing times
4. each tree from Step 3 is a separate SCC of  $G$

# SCC Algorithm Example



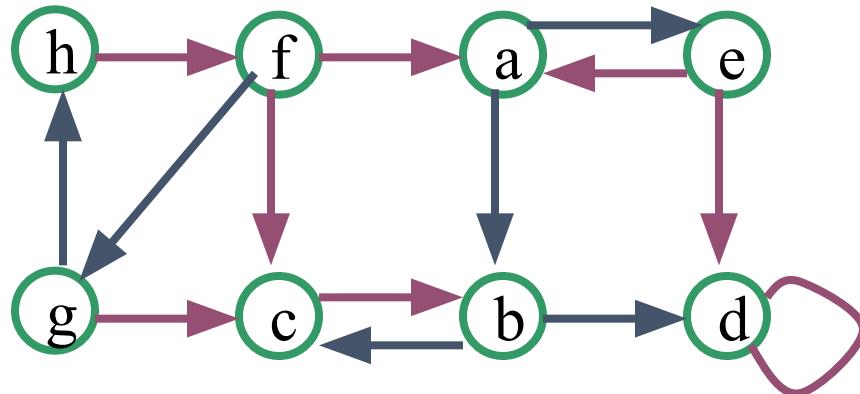
input graph - run DFS

# SCC Algorithm Example: After Step 1

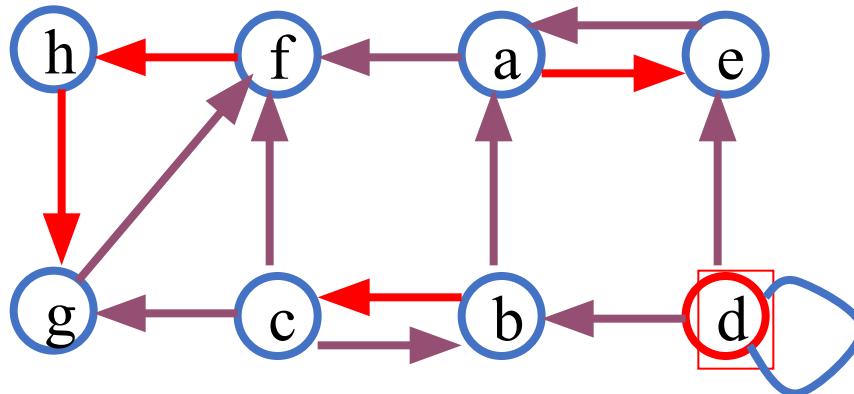


Order of vertices for Step 3: f, g, h, a, e, b, d, c

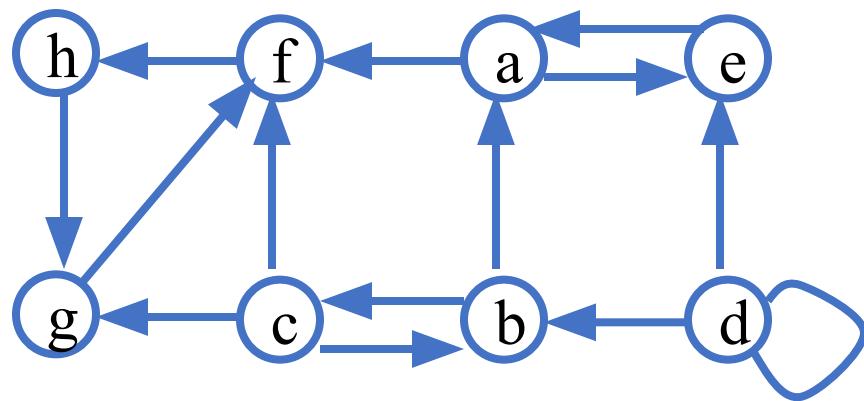
# SCC Algorithm Example: After Step 1



Order of vertices for Step 3: f, g, h, a, e, b, d, c



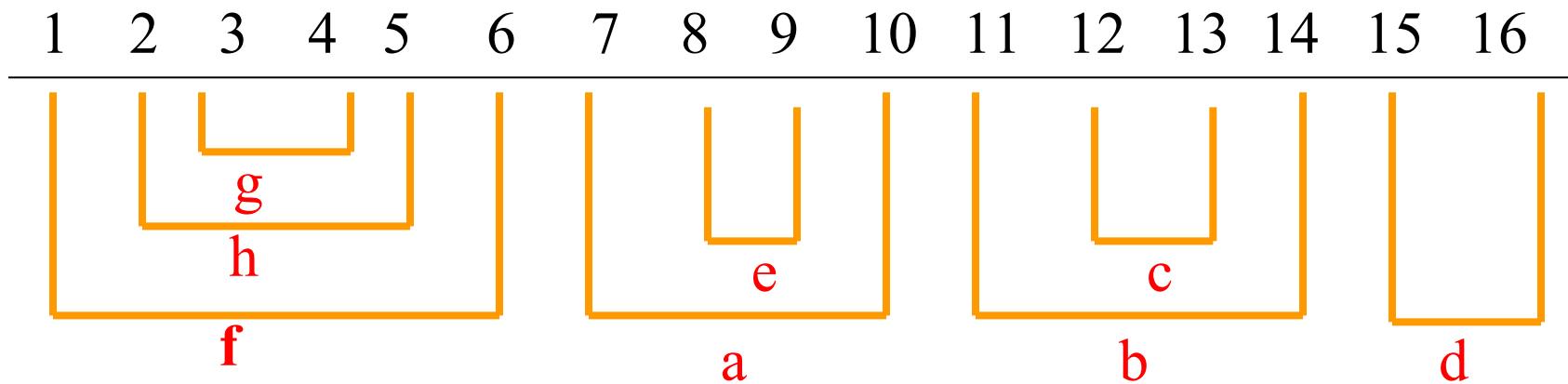
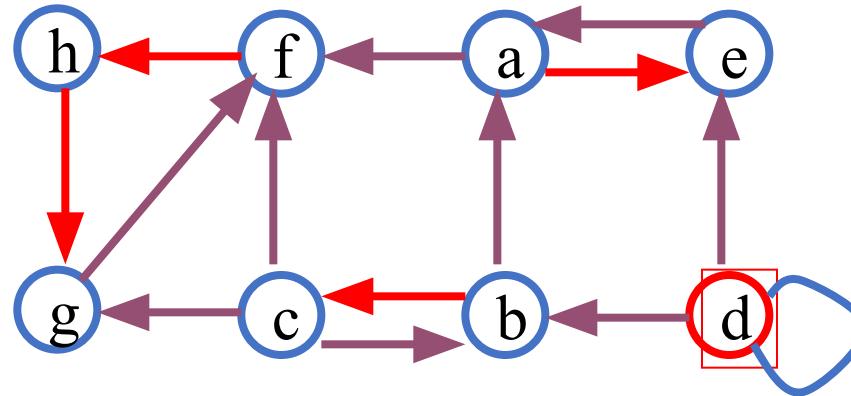
# SCC Algorithm Example: After Step 2



transposed input graph - run DFS with specified order of vertices

# SCC Algorithm Example: After Step 3

Order of vertices for Step 3: f, g, h, a, e, b, d, c



SCCs are  $\{f, h, g\}$ ,  $\{a, e\}$ ,  $\{b, c\}$ , and  $\{d\}$

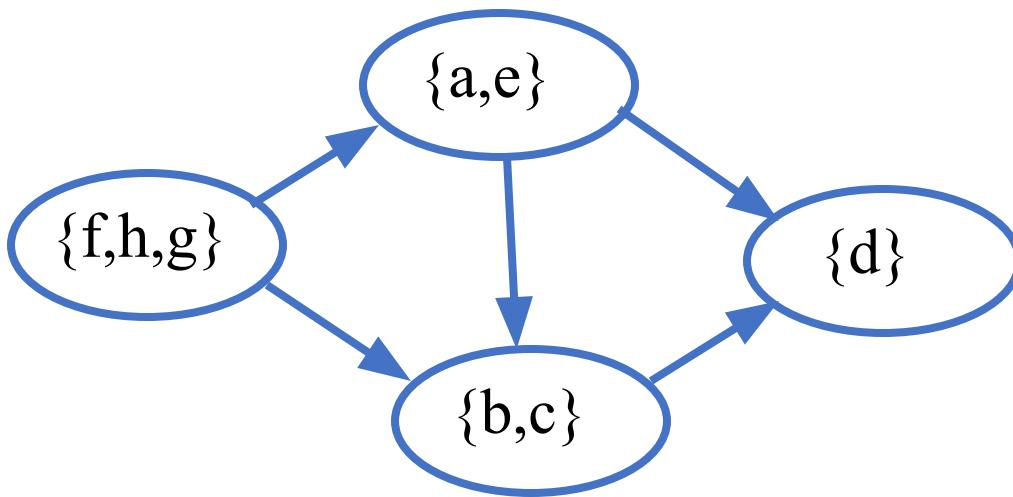
# Running Time of SCC Algorithm

- Step 1:  $O(V + E)$  to run DFS
- Step 2:  $O(V + E)$  to construct transpose graph,  
assuming adjacency list rep.
- Step 3:  $O(V + E)$  to run DFS again
- Step 4:  $O(V)$  to output result
- Total:  $O(V + E)$

# Correctness of SCC Algorithm

- Proof uses concept of **component graph**  $G^{SCC}$ , of  $G$ .
- Vertices are the SCCs of  $G$ ; call them  $C_1, C_2, \dots, C_k$
- Put an edge from  $C_i$  to  $C_j$  iff  $G$  has an edge from a vertex in  $C_i$  to a vertex in  $C_j$

# Example of Component Graph



based on example graph from before

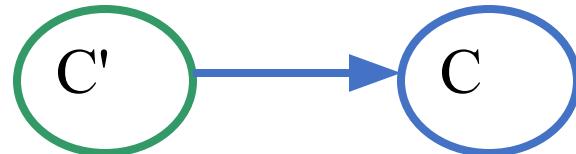
# Facts About Component Graph

- **Claim:**  $G^{SCC}$  is a directed acyclic graph.
- Why?
- Suppose there is a cycle in  $G^{SCC}$  such that component  $C_i$  is reachable from component  $C_j$  and vice versa.
- Then  $C_i$  and  $C_j$  would not be separate SCCs.

# Facts About Component Graph

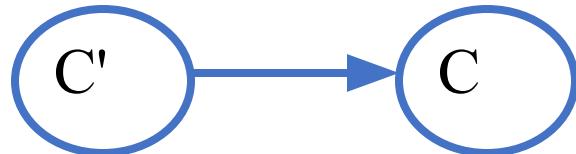
- Consider any component  $C$  during Step 1 (running DFS on  $G$ )
- Let  $d(C)$  be *earliest* discovery time of any vertex in  $C$
- Let  $f(C)$  be *latest* finishing time of any vertex in  $C$
- **Lemma:** If there is an edge in  $G^{SCC}$  from component  $C'$  to component  $C$ , then
$$f(C') > f(C).$$

# Proof of Lemma



- **Case 1:**  $d(C') < d(C)$ .
- Suppose  $x$  is first vertex discovered in  $C'$ .
- By the way DFS works, all vertices in  $C'$  and  $C$  become descendants of  $x$ .
- Then  $x$  is last vertex in  $C'$  to finish and finishes after all vertices in  $C$ .
- Thus  $f(C') > f(C)$ .

# Proof of Lemma



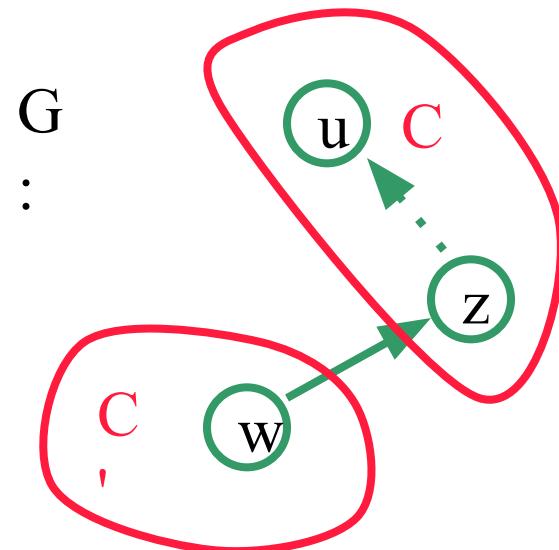
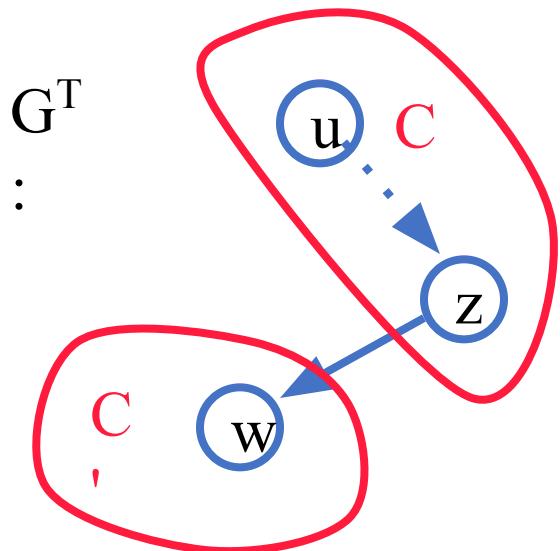
- **Case 2:**  $d(C') > d(C)$ .
- Suppose  $y$  is first vertex discovered in  $C$ .
- By the way DFS works, all vertices in  $C$  become descendants of  $y$ .
- Then  $y$  is last vertex in  $C$  to finish.
- Since  $C' \rightarrow C$ , no vertex in  $C'$  is reachable from  $y$ , so  $y$  finishes before any vertex in  $C'$  is discovered.
- Thus  $f(C') > f(C)$ .

# SCC Algorithm is Correct

- Prove this theorem by induction on number of trees found in Step 3 (running DFS on  $G^T$ ).
- Hypothesis is that the first  $k$  trees found constitute  $k$  SCCs of  $G$ .
- **Basis:**  $k = 0$ . No work to do !
- **Induction:** Assume the first  $k$  trees constructed in Step 3 (running DFS on  $G^T$ ) correspond to  $k$  SCCs; consider the  $(k+1)$ st tree.
- Let  $u$  be the root of the  $(k+1)$ st tree.
- $u$  is part of some SCC, call it  $C$ .
- By the inductive hypothesis,  $C$  is not one of the  $k$  SCCs already found and all so vertices in  $C$  are unvisited when  $u$  is discovered.
  - By the way DFS works, all vertices in  $C$  become part of  $u$ 's tree

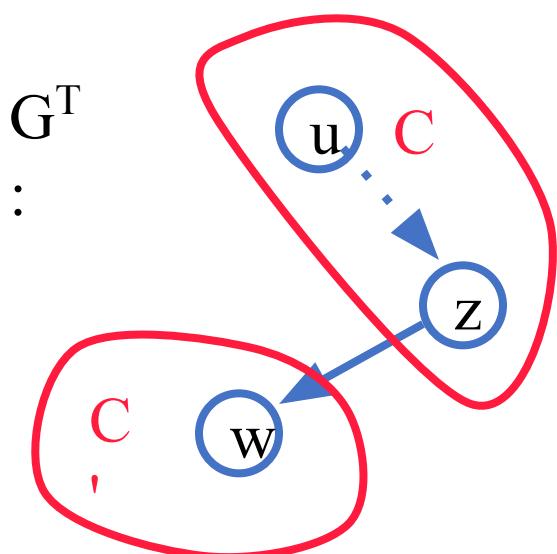
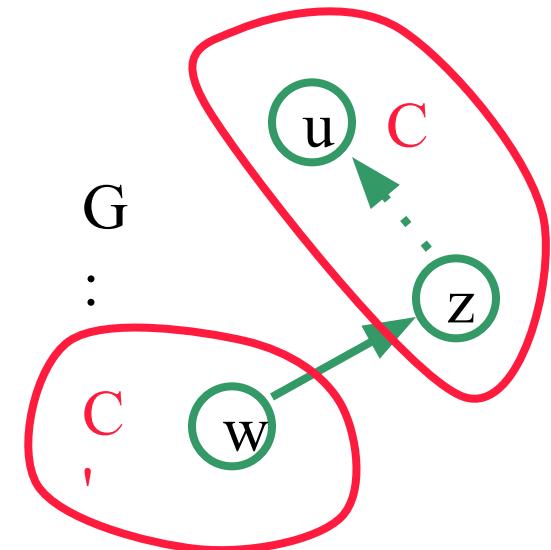
# SCC Algorithm is Correct

- Show *only* vertices in  $C$  become part of  $u$ 's tree.  
Consider an outgoing edge from  $C$ .



# SCC Algorithm is Correct

- By lemma, in Step 1 (running DFS on  $G$ ) the last vertex in  $C'$  finishes after the last vertex in  $C$  finishes [ $f(C') > f(C)$ ].
- Thus in Step 3 (running DFS on  $G^T$ ), some vertex in  $C'$  is discovered before any vertex in  $C$  is discovered.
- Thus in Step 3, all of  $C'$ , including  $w$ , is already visited before  $u$ 's DFS tree starts



*Thank  
you*

52

# CSE-215 Data Structures and Algorithm-II

## Greedy Method

Activity Selection Problem

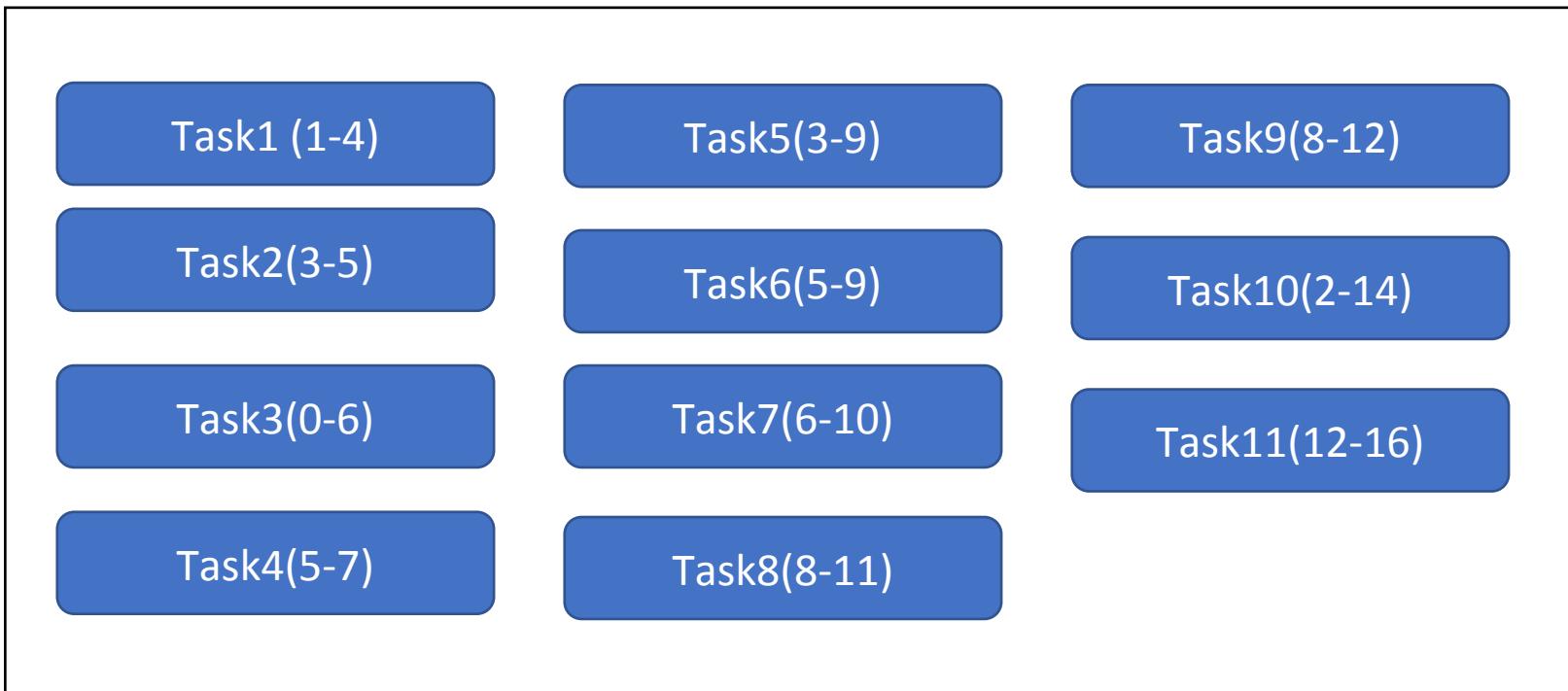
# The Activity Selection Problem

- **Input:** A set of activities  $S = \{a_1, a_2, \dots, a_n\}$ 
  - Each activity  $a_i$  has a start time  $s_i$  and a finish time  $f_i$ ,  
Where  $0 \leq s_i < f_i < \infty$
  - If selected, activity  $a_i$  takes place during the half-open time interval  $[s_i, f_i]$
  - Two activities are compatible if and only if their intervals do not overlap
- **Output:** a maximum-size subset of mutually compatible activities

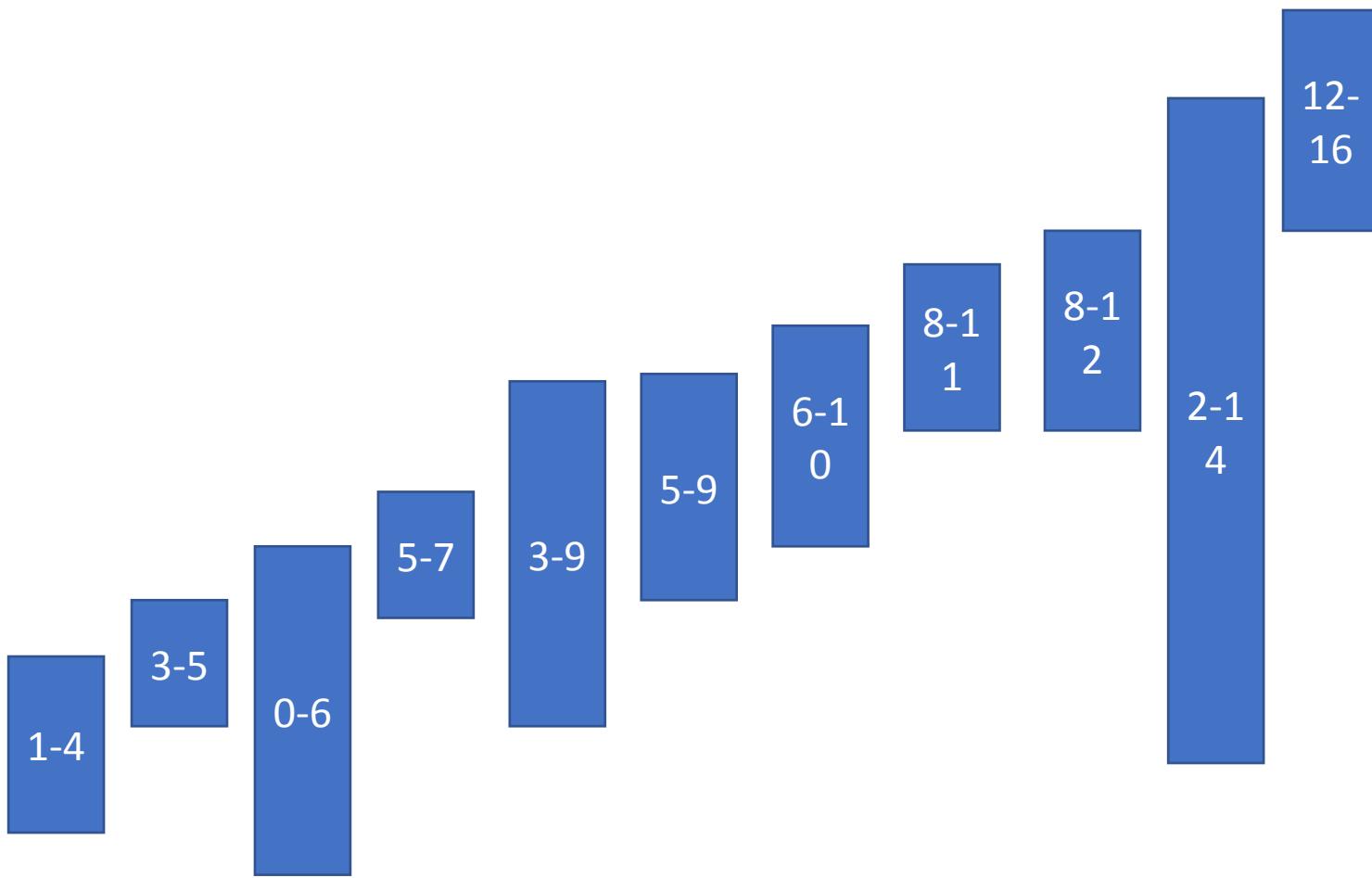
# The Activity Selection Problem

- *Formally:*
  - *Given a set  $S$  of  $n$  activities*
  - $s_i$  = *start time of activity  $i$*
  - $f_i$  = *finish time of activity  $I$*
- *Find max-size subset  $A$  of compatible activities*

# Lots of overlapping tasks



# Timeline



# Activity-Selection Problem Greedy Approach

i	1	2	3	4	5	6	7	8	9	10	11
start	1	3	0	5	3	5	6	8	8	2	12
end	4	5	6	7	9	9	10	11	12	14	16

- What is the maximum number of activities that can be completed?
- $\{a_3, a_9, a_{11}\}$  can be completed

# Activity-Selection Problem Greedy Approach

i	1	2	3	4	5	6	7	8	9	10	11
start	1	3	0	5	3	5	6	8	8	2	12
end	4	5	6	7	9	9	10	11	12	14	16

- What is the maximum number of activities that can be completed
- $\{a_3, a_9, a_{11}\}$  can be completed
- But so can  $\{a_1, a_4, a_8, a_{11}\}$  which is a larger set

# Activity-Selection Problem Greedy Approach

i	1	2	3	4	5	6	7	8	9	10	11
start	1	3	0	5	3	5	6	8	8	2	12
end	4	5	6	7	9	9	10	11	12	14	16

- What is the maximum number of activities that can be completed
  - $\{a_3, a_9, a_{11}\}$  can be completed
  - But so can  $\{a_1, a_4, a_8, a_{11}\}$  which is a larger set
  - But it is not unique, consider  $\{a_2, a_4, a_9, a_{11}\}$

# Solving Activity Selection Problem

The greedy choice can be applied to find solutions (the maximum number of activities performed) by using

- Earliest finish time
- Earliest start time
- Shortest interval

*Gives always optimal solution*

# Early Finish Greedy

- Select the activity with the earliest finish
- Eliminate the activities that could not be scheduled
- Repeat!

# Activity-Selection Problem Greedy Approach

i	1	2	3	4	5	6	7	8	9	10	11
start	1	3	0	5	3	5	6	8	8	2	12
end	4	5	6	7	9	9	10	11	12	14	16

This algorithm expects that the activities are sorted by end time

Need to create activity objects

# Activity-Selection Problem Greedy Approach

i	1	2	3	4	5	6	7	8	9	10	11
start	1	3	0	5	3	5	6	8	8	2	12
end	4	5	6	7	9	9	10	11	12	14	16

The diagram illustrates the greedy approach to solving the Activity-Selection Problem. It shows a list of 11 activities (i=1 to 11) with their start and end times. The activities are represented by rows in a table. The first row contains activity identifiers (i), and the subsequent rows contain their start and end times. Four specific activities are highlighted: Activity 1 (start=1, end=4), Activity 4 (start=5, end=7), Activity 8 (start=8, end=11), and Activity 11 (start=12, end=16). These highlighted activities are connected by blue arrows pointing upwards towards the table, indicating they are selected based on the greedy rule of choosing the activity that starts earliest among those that do not overlap with the currently selected activity.

i	1	2	3	4	5	6	7	8	9	10	11
start	1	3	0	5	3	5	6	8	8	2	12
end	4	5	6	7	9	9	10	11	12	14	16



## GREEDY-ACTIVITY-SELECTOR( $s, f$ )

```

1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 

```

# Time - Complexity

- $O(\log n)$  for the sorting
- $O(n)$  for traversing the objects
- Hence  $O(n \log n)$

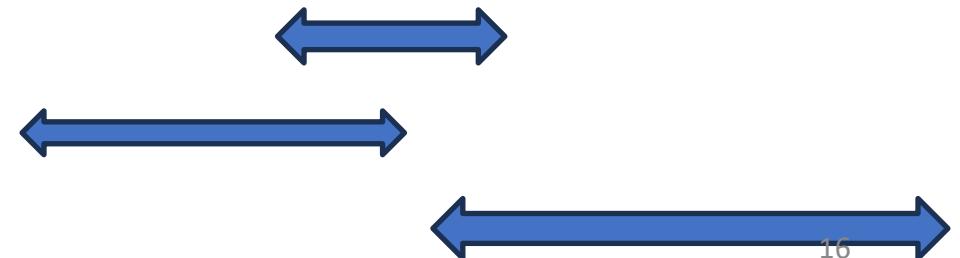
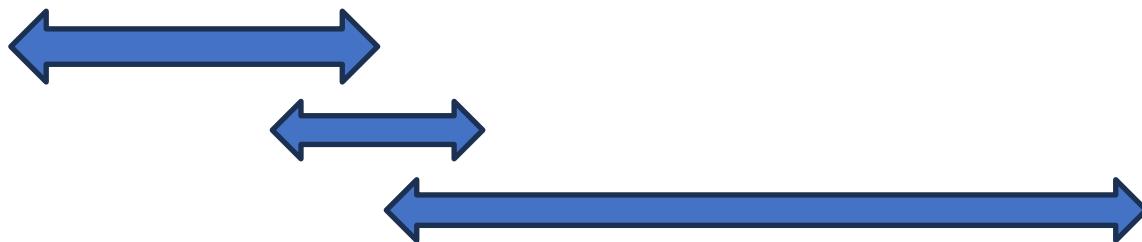
# Early Start Greedy

- Select the activity with the earliest start
- Eliminate the activities that could not be scheduled
- Repeat!

i	1	2	3	4	5	6	7	8	9	10	11
start	1	3	0	5	3	5	6	8	8	2	12
end	4	5	6	7	9	9	10	11	12	14	16

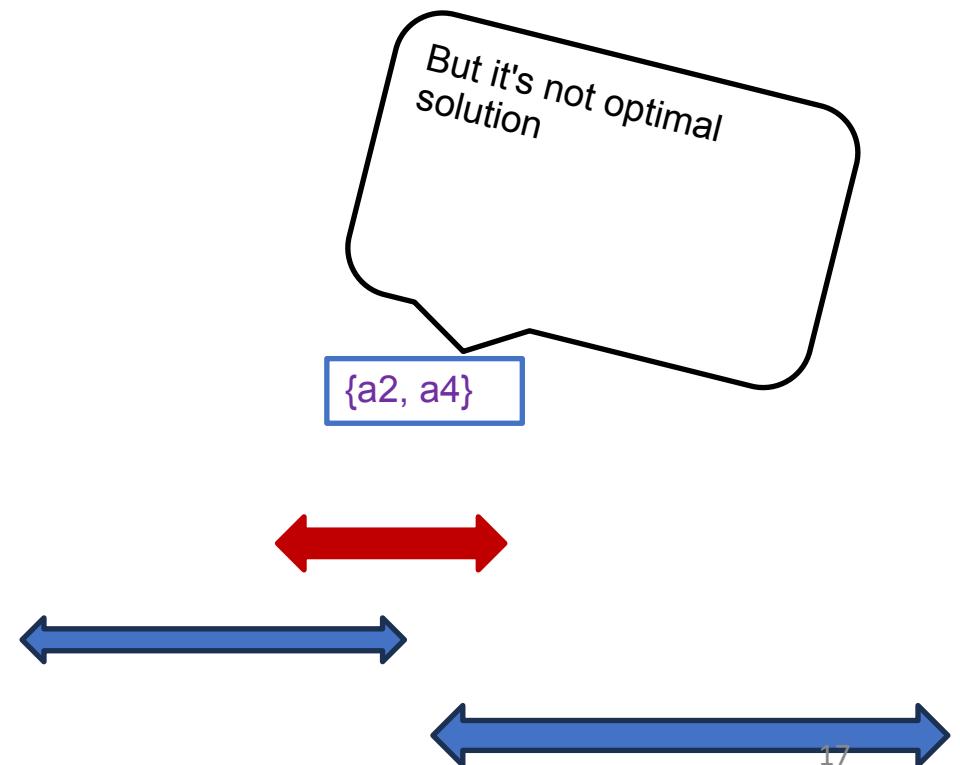
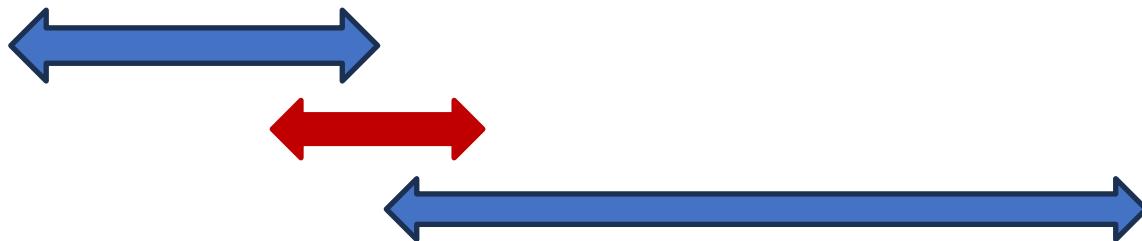
# Shortest Interval Greedy

- Select the activity with the **shortest interval**
- Eliminate the activities that could not be scheduled
- Repeat!



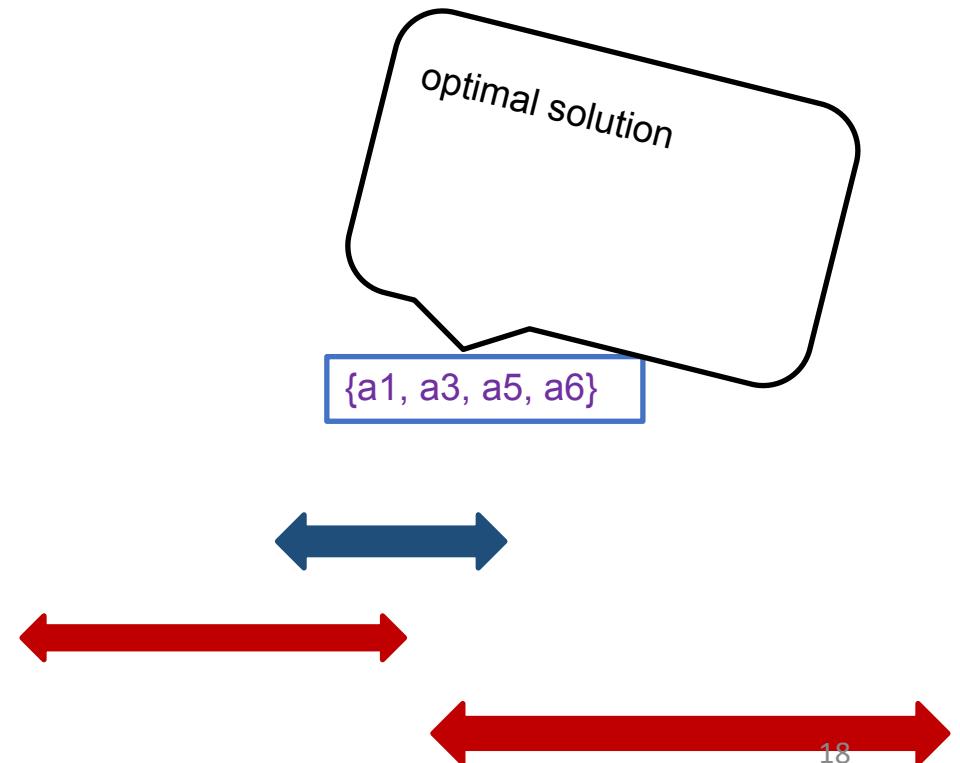
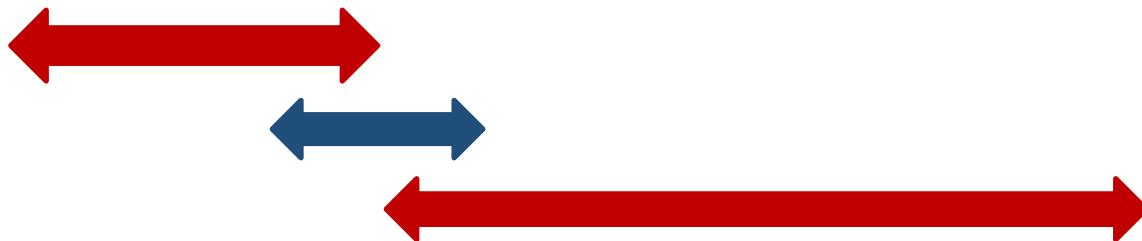
# Shortest Interval Greedy

- Select the activity with the **shortest interval**
- Eliminate the activities that could not be scheduled
- Repeat!



# Shortest Interval Greedy

- Select the activity with the **shortest interval**
- Eliminate the activities that could not be scheduled
- Repeat!



# Sample Question

Suppose that you run a car lending business, and each request is at the format: [start time, end time). A car can be used by one user at a time. Suppose that you have got the following car-use requests for the next day.

$$\{[1, 6), [4, 8), [6, 8), [9, 15), [6, 9), [11, 15), [2, 6)\}$$

Find **all solutions** of the maximum number of requests that can be satisfied by using

- earliest end time,
- shortest interval,
- earliest start time.

Thank  
you

20

# *Divide-and-Conquer Technique: Solving Recurrences*

# How to Solve Recurrence Relations?

Methods for solving recurrence relations:

- iteratively expansion
- using the recursion tree
- Master Theorem

# Solving Recurrences: Iterative Expansion

Finding Maximum and Minimum:

$$T(n) = 2T(n/2) + b$$

$$= 2 [ 2T(n/4) + b ] + b$$

$$= 2^2 T(n/2^2) + 2b + b$$

$$= 2^2 [ 2 T(n/8) + b ] + 2b + b$$

$$= 2^3 T(n/2^3) + 2^2b + 2b + b$$

= .....

$$= 2^i T(n/2^i) + 2^i b + ..... + 2^2b + 2b + b$$

$$= 2^i T(n/2^i) + (2^{i-1+1} - 1) b$$

$$T(n) = \begin{cases} b & \text{if } n = 1 \text{ or } 2 \\ 2T(n/2) + b & \text{if } n > 2 \end{cases}$$

We stop when  $n/2^i = 1$ . That is  $2^i = n$

Then,  $i = \log_2 n$

So,  $T(n) = n T(1) + b(n - 1) = nb + b(n - 1) = 2bn - 1$

Thus,  $T(n) = O(n)$ .

# Solving Recurrences: Iterative Expansion

Merge Sort (both Best case and Worst case):

$$\begin{aligned} T(n) &= 2T(n/2) + bn \\ &= 2[2T(n/4) + b(n/2)] + bn \\ &= 2^2 T(n/2^2) + bn + bn \\ &= 2^2 [2T(n/8) + b(n/4)] + 2bn \\ &= 2^3 T(n/2^3) + 3bn \\ &= 2^4 T(n/2^4) + 4bn \\ &= \dots\dots\dots \\ &= 2^i T(n/2^i) + ibn \end{aligned}$$

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n > 1 \end{cases}$$

We stop when  $n/2^i = 1$ . That is  $2^i = n$

Then,  $i = \log_2 n$

So,  $T(n) = n T(1) + \log_2 n (bn) = bn + bn\log_2 n$

Thus,  $T(n) = O(n\log_2 n)$

# Solving Recurrences: Iterative Expansion

Quick Sort (Best case):

$$T(n) = 2T(n/2) + bn$$

$$= 2 [ 2T(n/4) + b(n/2) ] + bn$$

$$= 2^2 T(n/2^2) + bn + bn$$

$$= 2^2 [ 2 T(n/8) + b(n/4) ] + 2bn$$

$$= 2^3 T(n/2^3) + 3bn$$

$$= 2^4 T(n/2^4) + 4bn$$

= .....

$$= 2^i T(n/2^i) + ibn$$

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n > 1 \end{cases}$$

We stop when  $n/2^i = 1$ . That is  $2^i = n$

Then,  $i = \log_2 n$

So,  $T(n) = n T(1) + \log_2 n (bn) = bn + bn \log_2 n$

Thus,  $T(n) = O(n \log_2 n)$

# Solving Recurrences: Iterative Expansion

Quick Sort (Worst case):

$$\begin{aligned} T(n) &= T(n-1) + bn \\ &= T[(n-2) + b(n-1)] + bn \\ &= T[(n-3) + b(n-2)] + b(n-1) + bn \\ &= T[(n-4) + b(n-3)] + b(n-2) + b(n-1) + bn \\ &= \dots \\ &= T(n-i) + b[(n-i+1) + \dots + (n-2) + (n-1) + n] \end{aligned}$$

$$T(n) = \begin{cases} b & \text{if } n = 1 \\ T(n-1) + bn & \text{if } n > 1 \end{cases}$$

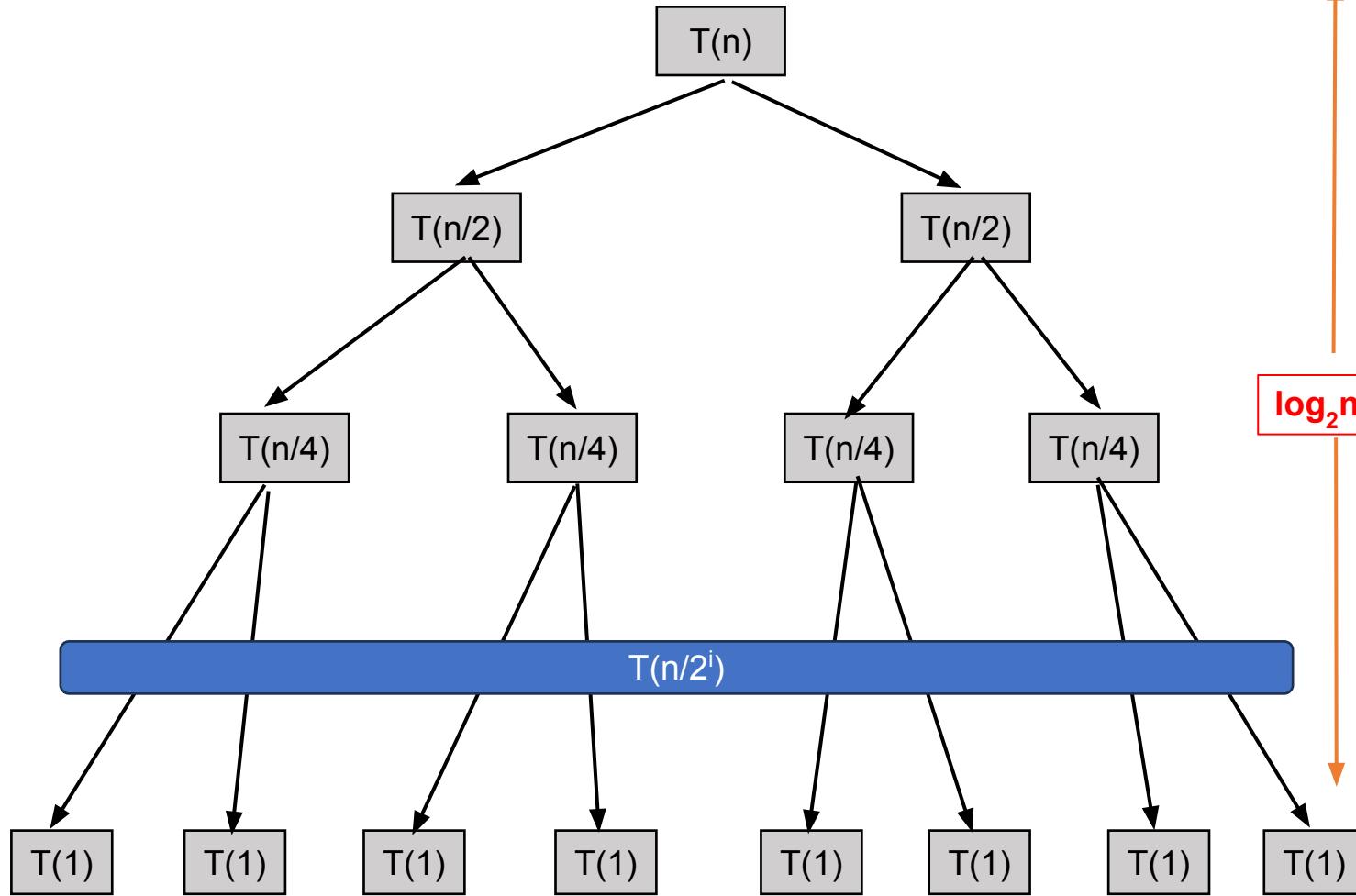
We stop when  $n - i = 1$ . That is  $i = n-1$

$$\begin{aligned} \text{Then, } T(n) &= T(1) + b[2 + \dots + (n-2) + (n-1) + n] \\ &= b + b[2 + \dots + (n-2) + (n-1) + n] \\ &= b[1 + 2 + \dots + (n-2) + (n-1) + n] \\ &= b[n(n+1)/2] \end{aligned}$$

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

# Solving Recurrences: Recursion Tree

Merge Sort (both Best case and Worst case):



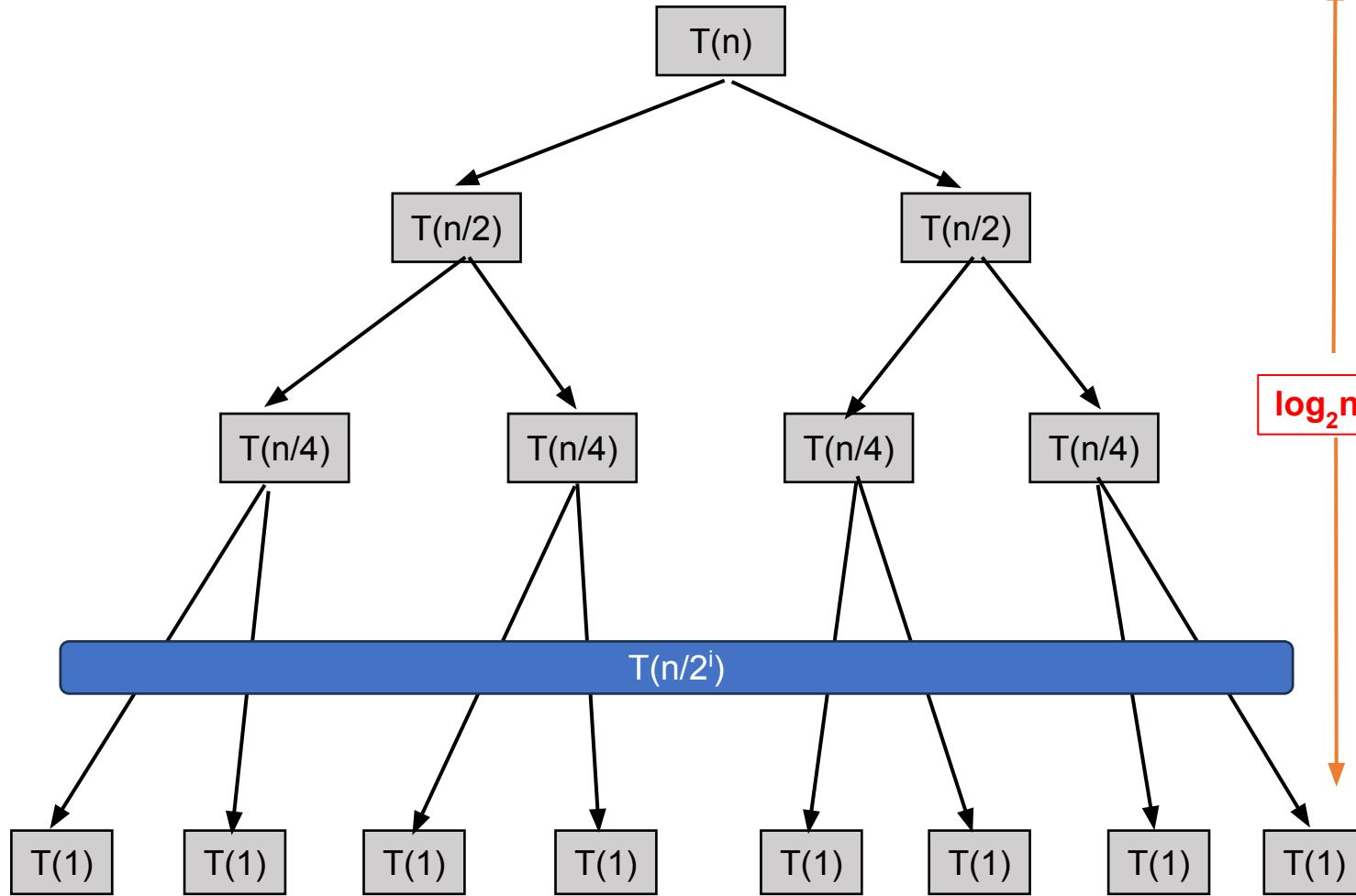
$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n > 1 \end{cases}$$

No. of Nodes at this Level	Time for Each Node	Time at this Level
1	$bn$	$1(bn) = bn$
2	$b(n/2)$	$2(bn/2) = bn$
4	$b(n/2^2)$	$4(bn/4) = bn$
$2^i$	$b(n/2^i)$	$2^i(bn/2^i) = bn$
$n$	$b$	$n(b) = bn$

$$T(n) = bn + bn + \dots + bn = bn \log n = O(n \log n)$$

# Solving Recurrences: Recursion Tree

Finding Maximum and Minimum:



$$T(n) = \begin{cases} b & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n > 1 \end{cases}$$

No. of Nodes at this Level	Time for Each Node	Time at this Level
1	b	$1(b) = b$
2	b	$2(b) = 2b$
4	b	$4(b) = 4b$
$2^i$	b	$2^i(b) = 2^i b$
n	b	$n(b) = bn$

$$T(n) = b + 2b + 2^2b + \dots + 2^i b + \dots + bn = 2bn - 1 = O(n)$$

# Master Theorem: Solving Recurrences

# The Master Theorem

- Given: a *divide and conquer* algorithm
  - An algorithm that divides the problem of size  $n$  into  $a$  subproblems, each of size  $n/b$ , where  $a \geq 1, b > 1$
  - The  $a$  subproblems are solved recursively, each in time  $T(n/b)$
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function  $f(n)$ , where  $f$  is asymptotically positive
- Then, the Master Theorem gives us a **cookbook** for the algorithm's running time.

# The Master Theorem

Let  $a \geq 1$  and  $b > 1$  be constants, let  $f(n)$  be a function, and let  $T(n)$  be defined on the nonnegative integers by the recurrence

$$T(n) = a T(n/b) + f(n).$$

Then  $T(n)$  has the following asymptotic bounds:

- If  $f(n) = O(n^{\log_b a - \varepsilon})$  for some constant  $\varepsilon > 0$ , then

$$T(n) = \Theta(n^{\log_b a})$$

- If  $f(n) = \Theta(n^{\log_b a})$  then

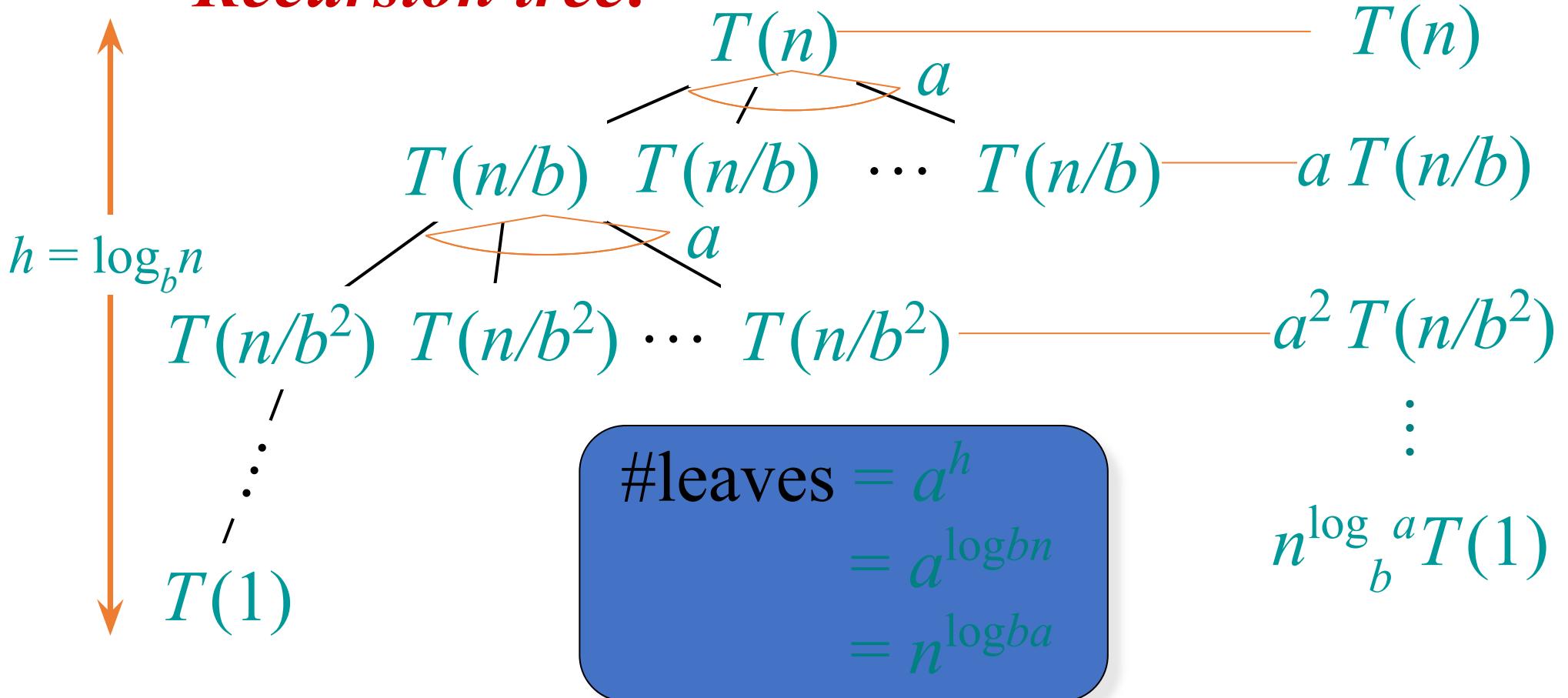
$$T(n) = \Theta(n^{\log_b a} \log n)$$

- If  $f(n) = O(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$  then

$$T(n) = \Theta(f(n))$$

# Idea of master theorem

*Recursion tree:*



## Three common cases

Compare  $f(n)$  with  $n^{\log ba}$ :

1.  $f(n) = O(n^{\log ba - \varepsilon})$  for some constant  $\varepsilon > 0$ .
  - $f(n)$  grows polynomially slower than  $n^{\log ba}$  (by an  $n^\varepsilon$  factor).

***Solution:***  $T(n) = \Theta(n^{\log ba})$ .

## Three common cases

Compare  $f(n)$  with  $n^{\log ba}$ :

2.  $f(n) = \Theta(n^{\log ba})$ .

- $f(n)$  and  $n^{\log ba}$  grow at similar rates.

**Solution:**  $T(n) = \Theta(n^{\log ba} \lg n)$  .

## Three common cases

Compare  $f(n)$  with  $n^{\log ba}$ :

3.  $f(n) = \Omega(n^{\log ba + \varepsilon})$  for some constant  $\varepsilon > 0$ .
  - $f(n)$  grows polynomially faster than  $n^{\log ba}$  (by an  $n^\varepsilon$  factor),

**Solution:**  $T(n) = \Theta(f(n))$  .

# Solving Recurrences: Master Theorem

Essentially, the Master theorem compares the function  $f(n)$  with the function  $g(n) = n^{\log_b a}$

Roughly, the theorem says:

If  $f(n) \ll g(n)$  then  $T(n) = O(g(n))$

If  $f(n) \sim g(n)$  then  $T(n) = O(g(n) \log_b n)$

If  $f(n) \gg g(n)$  then  $T(n) = O(f(n))$

# Examples

**Ex.**  $T(n) = 4T(n/2) + n$   
 $a = 4, b = 2 \Rightarrow n^{\log ba} = n^2; f(n) = n.$   
CASE 1:  $f(n) = O(n^{2-\varepsilon})$  for  $\varepsilon = 1$ .  
 $\therefore T(n) = \Theta(n^2).$

**Ex.**  $T(n) = 4T(n/2) + n^2$   
 $a = 4, b = 2 \Rightarrow n^{\log ba} = n^2; f(n) = n^2.$   
CASE 2:  $f(n) = \Theta(n^2 \lg n).$   
 $\therefore T(n) = \Theta(n^2 \lg n).$

## Examples

**Ex.**  $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \Rightarrow n^{\log ba} = n^2; f(n) = n^3.$

CASE 3:  $f(n) = \Omega(n^{2+\varepsilon})$  for  $\varepsilon = 1$

$\therefore T(n) = \Theta(n^3).$

# Master Theorem: Sample Question

*Solve the following recurrences by using the Master theorem.*

$$(i) T(n) = 4T(n/2) + 8n + 5$$

$$(ii) T(n) = 9T(n/3) + 5n\sqrt{n} + 8\log n$$

$$(iii) T(n) = 7T(n/4) + 3n^2 + 4n + 5$$

$$(iv) T(n) = 20T(n/4) + 3n^2 + 4n + 5$$

$$(v) T(n) = 9T(n/3) + 5n^2 + 7$$

$$(vi) T(n) = 6T(n/3) + 3n^2 + 7n$$

$$(vii) T(n) = 8T(n/2) + 5n^3 + 7n^2$$

Thank  
you

# NP Completeness

# Contents

1. Introduction and the 6 classes of problems
2. The P class and NP class
3. NP Complete Problems
4. NP-Hard, Co-P and Co-NP classes.
5. Difficulty levels of the problem classes and coping mechanisms
6. Approximation Algorithms

# 1. Introduction and the 6 classes of problems

Introductory Stuffs

Introduction of the 6 classes and the relation in between the classes

# Introduction

Some of the algorithms we have covered so far:

- Dynamic Programming
- Divide and Conquer
- Greedy

# The 6 classes of problems:

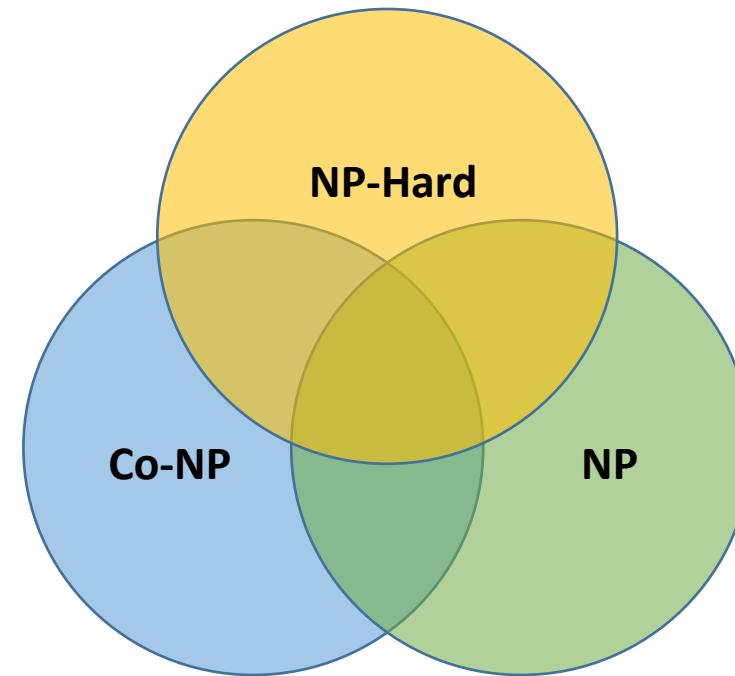
- Sets of problems:
  - Set A : {Job Scheduling, Activity Selection, ...}
  - Set B : {Merge Sort, Quick Sort, ...}
  - Set C : {BFS, DFS, ...}
  - ...
  - Set N : {ProbX, Prob Y}
- Class P = {SetA, SetB, SetC,..., SetN}

# The 6 classes of problems:

1. P
2. NP
3. NP Complete
4. NP Hard
5. Co-P
6. Co- NP

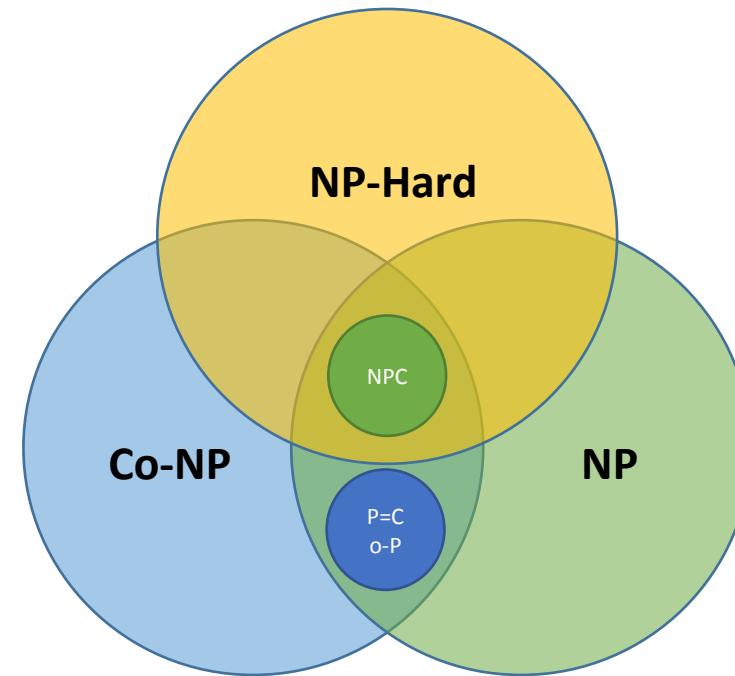
# The 6 classes of problems:

1. P
2. NP
3. NP Complete
4. NP Hard
5. Co-P
6. Co- NP



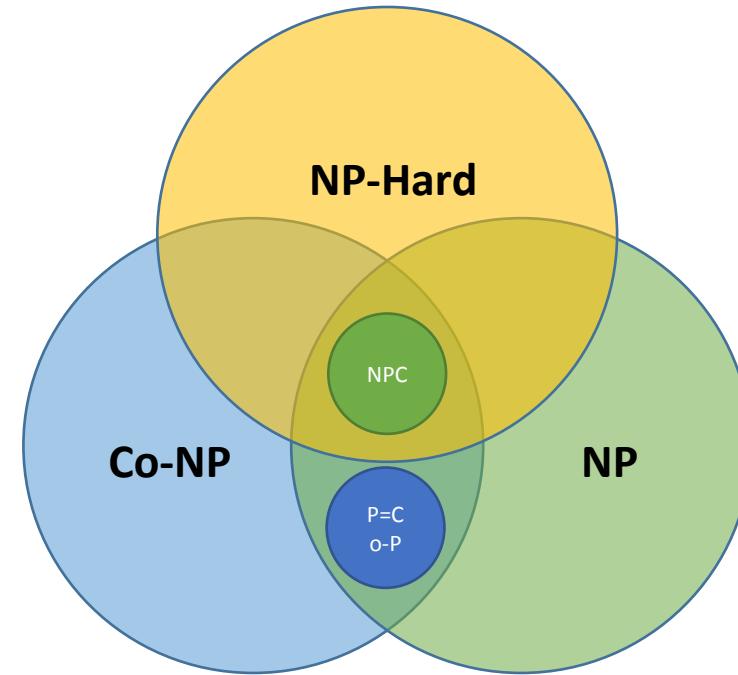
# The 6 classes of problems:

1. P
2. NP
3. NP Complete
4. NP Hard
5. Co-P
6. Co- NP



# The 6 classes of problems:

1. P
2. NP
3. NP Complete
4. NP Hard
5. Co-P
6. Co- NP



Most of the Computer  
Scientists accept this diagram

## 2. The P class and NP class

The P Class of Problems

The NP Class of Problems

A problem similar to an NP class problem, but is actually in P class

# P-Class of problems

P is a complexity class that represents the set of all **decision problems** that can be solved in **reasonable time**.

What constitutes reasonable time?

- Standard working definition: *polynomial time*
- On an input of size  $n$  the worst-case running time is  $O(n^k)$  for some constant  $k$
- **Polynomial time:**  $O(n^2)$ ,  $O(n^3)$ ,  $O(1)$ ,  $O(n \lg n)$
- **Not in polynomial time:**  $O(2^n)$ ,  $O(n^n)$ ,  $O(n!)$

# P Class of Problems (Cont.)

- Some problems are solvable in polynomial time.
  - Almost every algorithm we have studied so far.
  - These algorithms fall into the class P
- But not all problems are solvable in polynomial time.
  - We will see such problems soon.

# NP-Class of problems

**NP (Nondeterministic Polynomial time)** is the set of problems

- that **cannot be solved** in polynomial time **at the moment**
- A NP class problem will become a class P problem if someone **discovers** to solve it in **polynomial time**.
- However, if someone gives us an **NP problem's solution**, it can be **verified** in **polynomial time**.
- Hence, most of the problems which are in the **NP class**, are addressed as **verification problems**.
- Yes, we need an example to understand this.

# NP-Class of problems

An example of NP class problem:

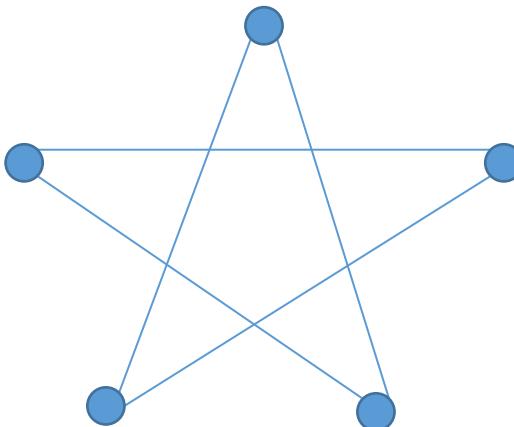
## **The Hamiltonian Cycle Problem.**

- Hamiltonian cycle is a circular path that covers **every vertex once**, and returns to the **first vertex**.
- A graph  $G=\{V,E\}$  is given.
- The problem requires us to determine whether there is a Hamiltonian cycle or not.

# NP-Class of problems

Hamiltonian cycle is a circular path that covers **every vertex once**, and returns to the **first vertex**.

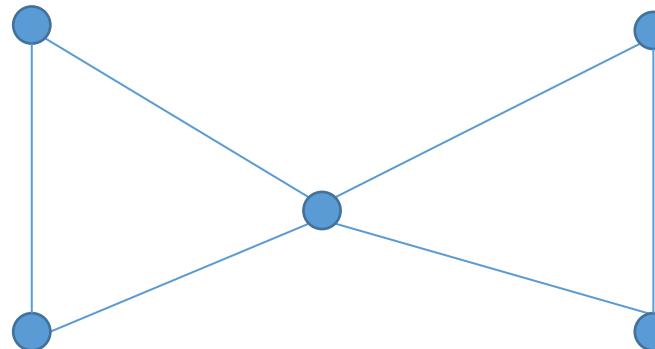
Is there a Hamiltonian cycle in this graph?



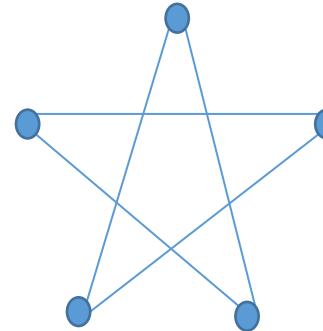
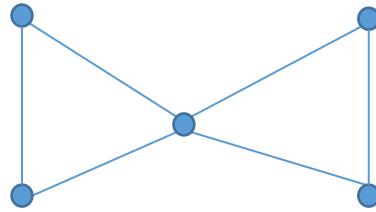
# NP-Class of problems

Hamiltonian cycle is a circular path that covers **every vertex once**, and returns to the **first vertex**.

Is there a Hamiltonian cycle in this graph?



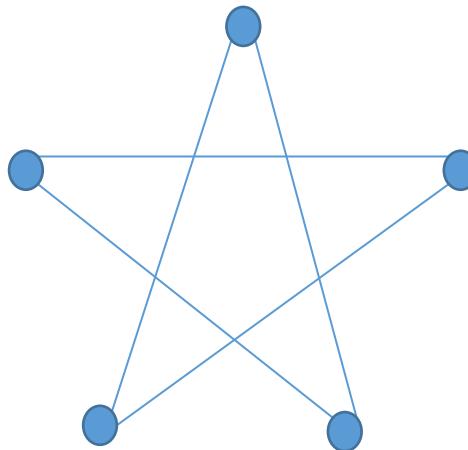
# NP-Class of problems



- These graphs were small.
- Please note, that there is **no polynomial time** algorithm, to **determine whether** a graph has a **Hamiltonian cycle** in it or **not**.

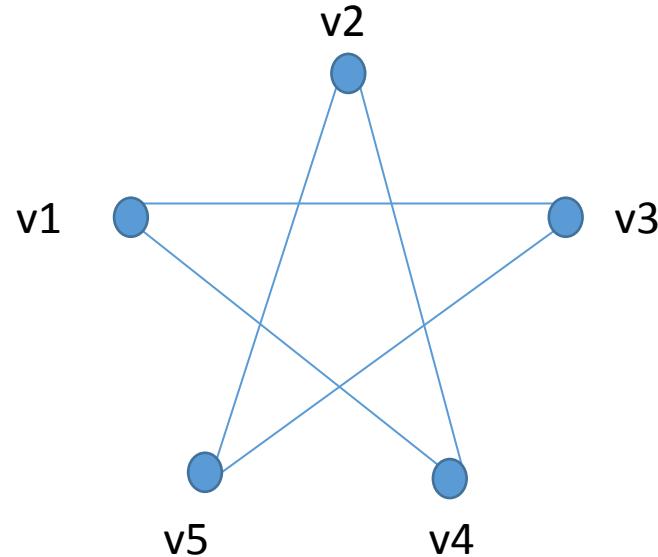
# NP-Class of problems

- However, if some one gives you the Hamiltonian cycle (i.e. the sequence of vertices, you can verify it in polynomial time)



# NP-Class of problems

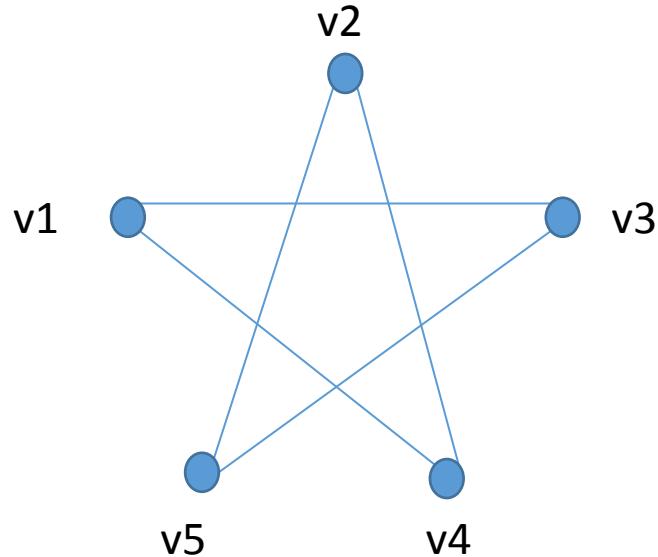
- However, if some one gives you the Hamiltonian cycle (i.e. the sequence of vertices, you can verify it in polynomial time)



$\{v1, v3, v5, v2, v4, v1\}$

# NP-Class of problems

- However, if some one gives you the Hamiltonian cycle (i.e. the sequence of vertices, you can verify it in polynomial time)

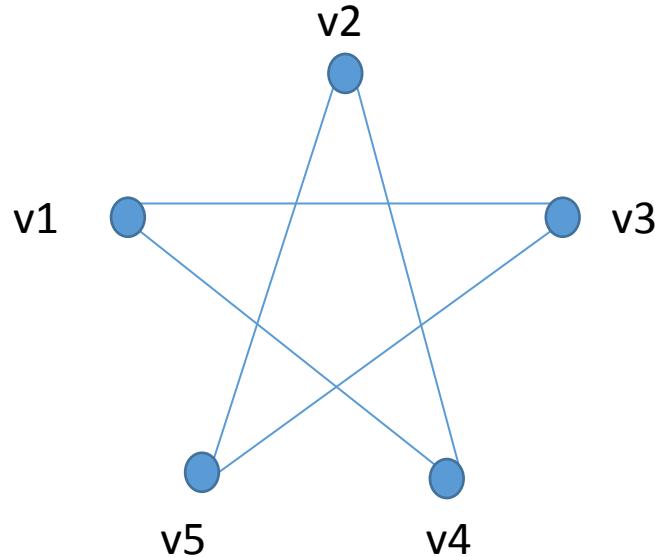


{v1,v3,v5,v2,v4,v1}

We can go iterate through the sequence, and check the Adj matrix to see if v1 is connected to v3, v3 is connected to v5 and so on.

# NP-Class of problems

- However, if some one gives you the Hamiltonian cycle (i.e. the sequence of vertices, you can verify it in polynomial time)



$\{v1, v3, v5, v2, v4, v1\}$

THUS WE CAN VERIFY IT!

# A similar problem , but in P class.

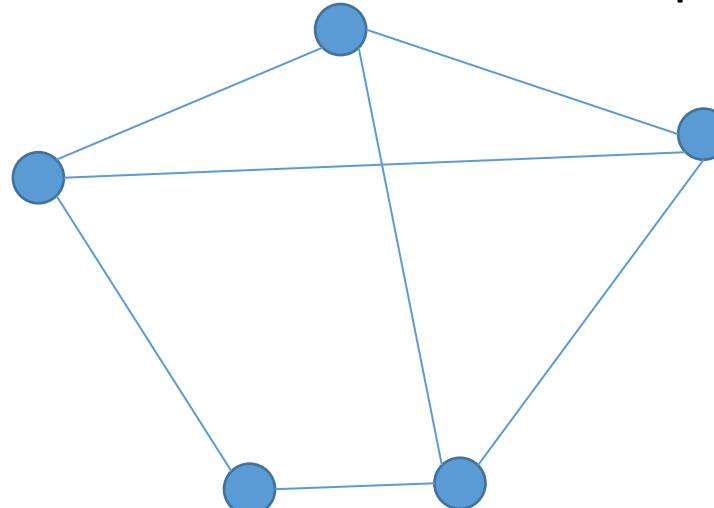
## Euler tour:

- A graph is given .
- The journey is started from the first vertex and it ends at the first vertex.
- Each **edge** is traversed **only once**.
- The problem:
  - We need to determine whether Euler tour is possible in the graph or not

# A similar problem , but in P class.

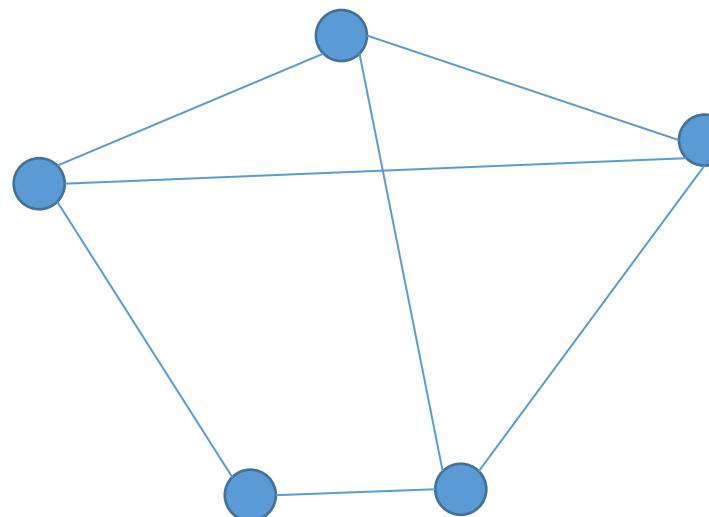
## Euler tour:

- A graph is given .
- The journey is started from the first vertex and it ends at the first vertex.
- Each **edge** is traversed **only once**.
- The problem:
  - We need to determine whether Euler tour is possible in the graph or not



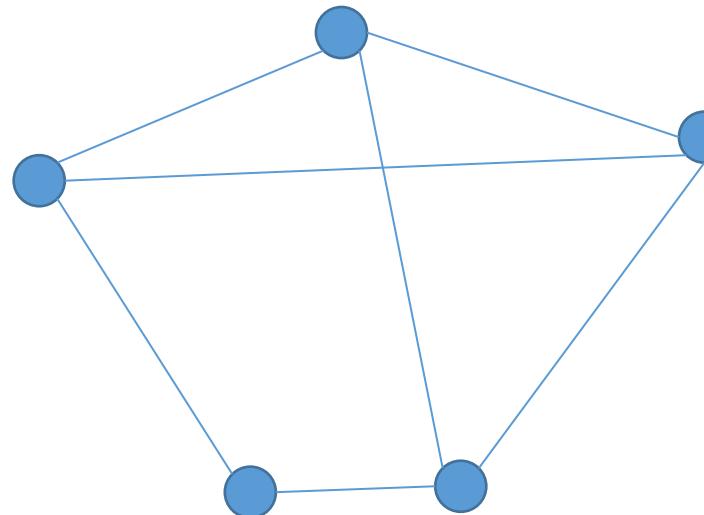
A similar problem , but in P class.

**Do you think there is a polynomial time algorithm, to determine whether a graph has Euler tour or not?**



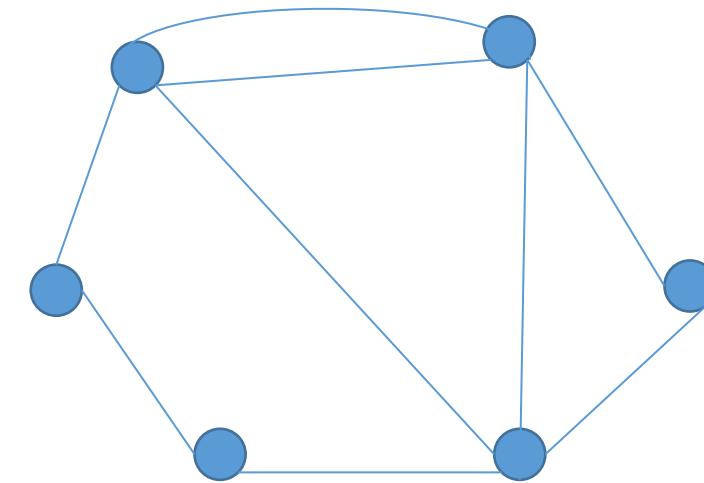
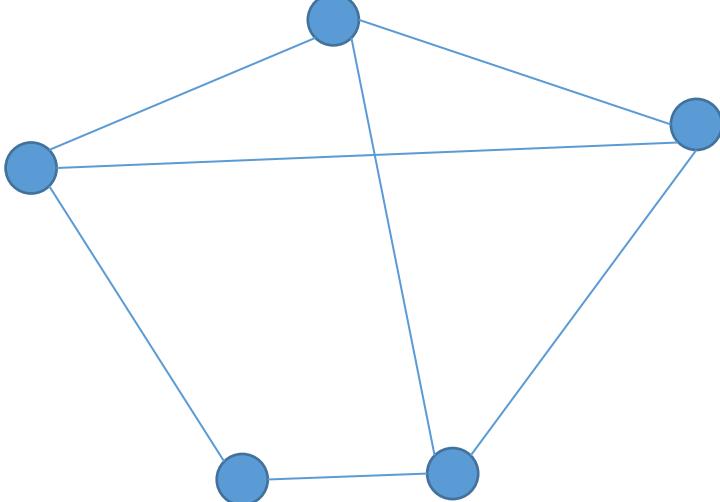
A similar problem , but in P class.

Do you think there is a polynomial time algorithm, to determine whether a graph has Euler tour or not?



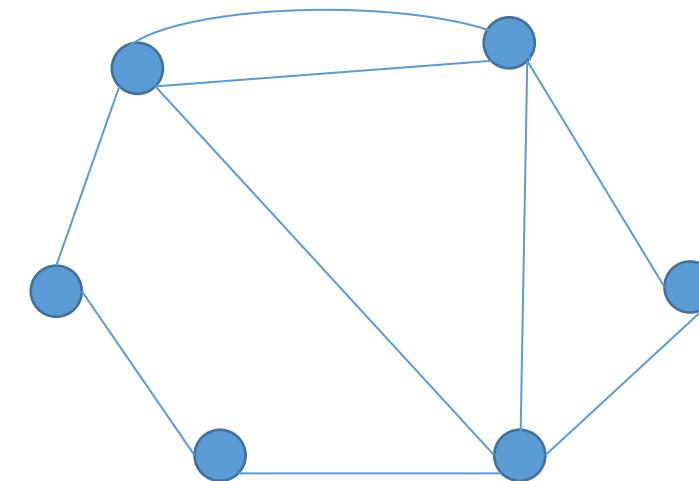
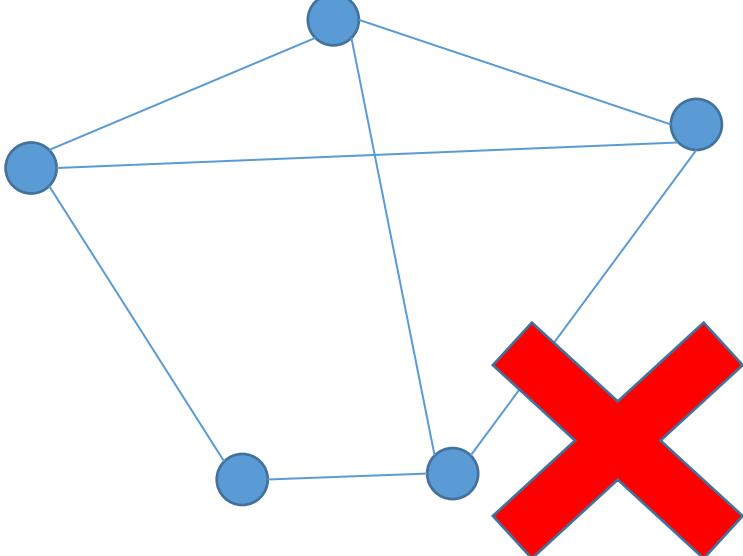
# A similar problem , but in P class.

Iterate through, each vertex, if the **degree** of each vertex is **even**, then Euler tour is possible.



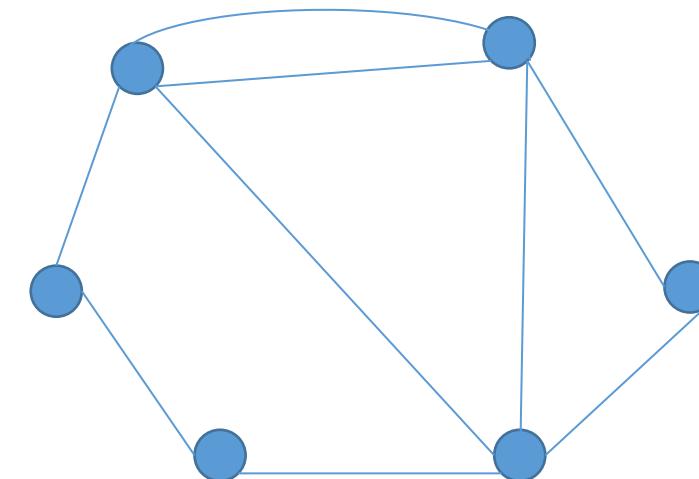
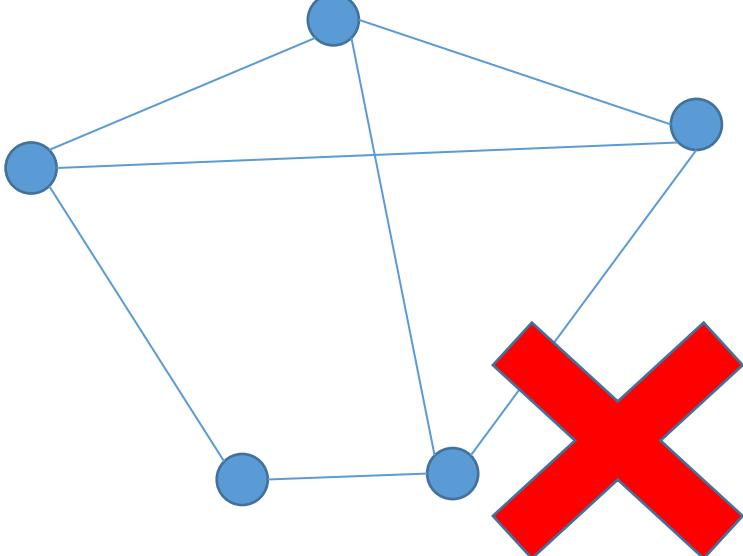
# A similar problem , but in P class.

Iterate through, each vertex, if the **degree** of each vertex is **even**, then Euler tour is possible.



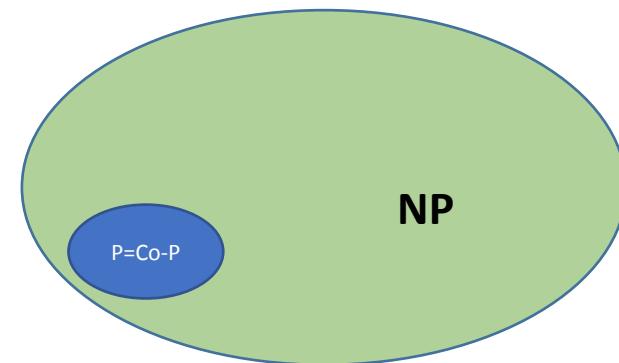
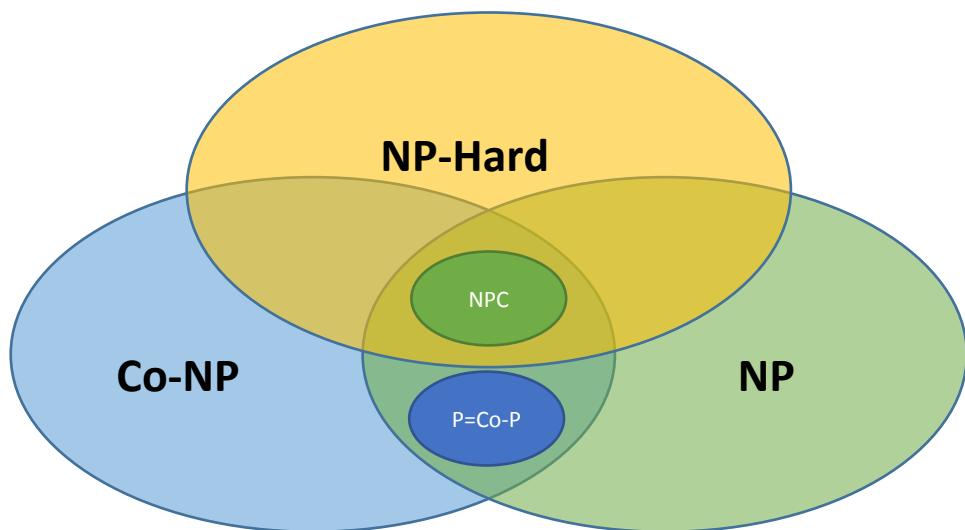
# A similar problem , but in P class.

Iterate through, each vertex, if the **degree** of each vertex is **even**, then Euler tour is possible.



# A similar problem , but in P class.

- Note that, problems in P class are not only **solvable** in polynomial time, but also the **solutions** of these problems are **verifiable** in polynomial time.
- This is the reason why the P class problems are within NP class.



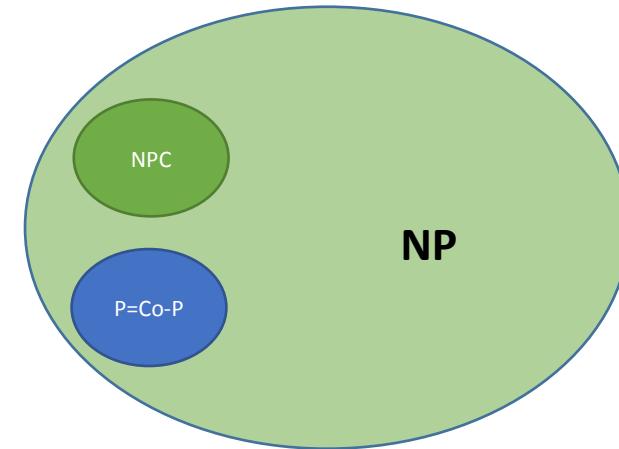
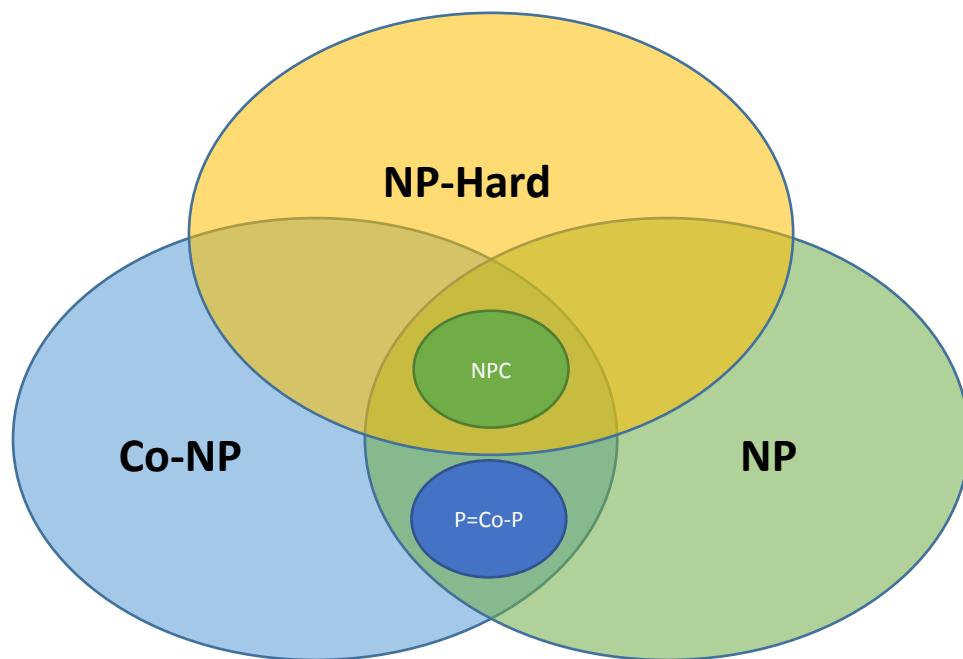
# 3. NP Complete Problems

What is an NP Complete Problem?

Why are NP complete Problems important?

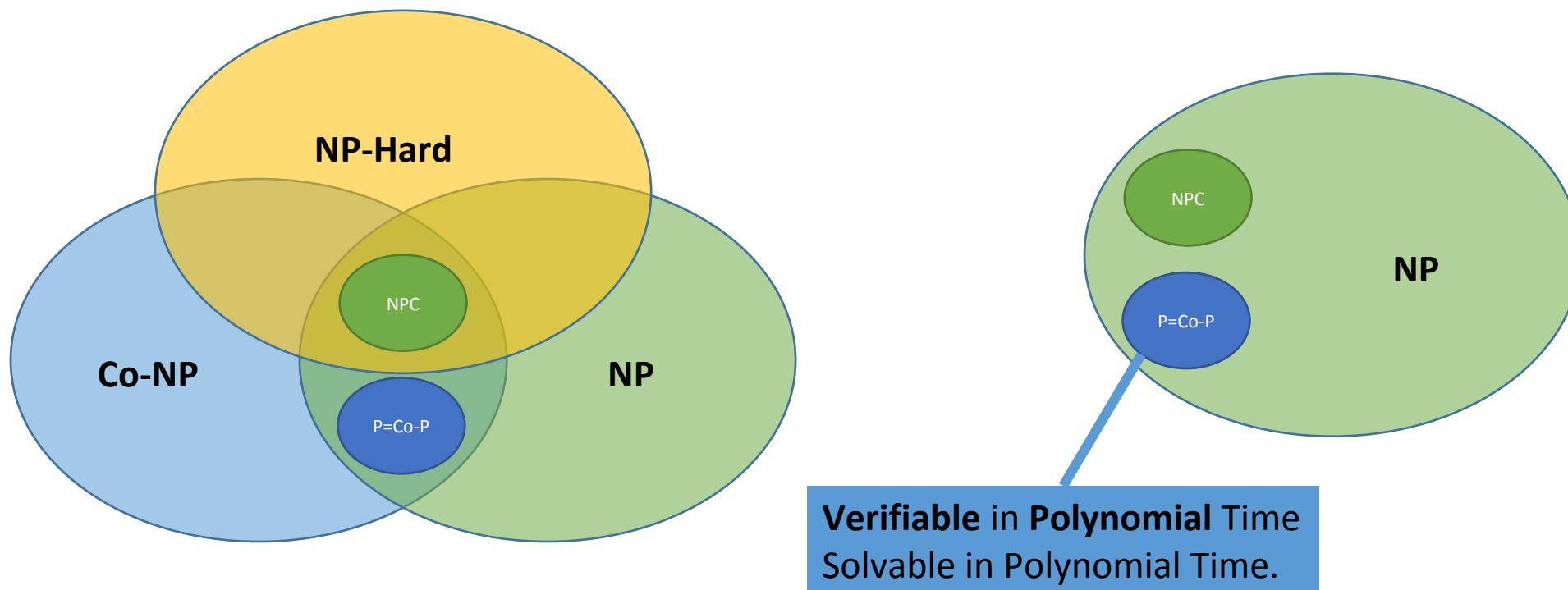
# NP Complete Problems

Let us take a look at the problems class Venn diagram again:



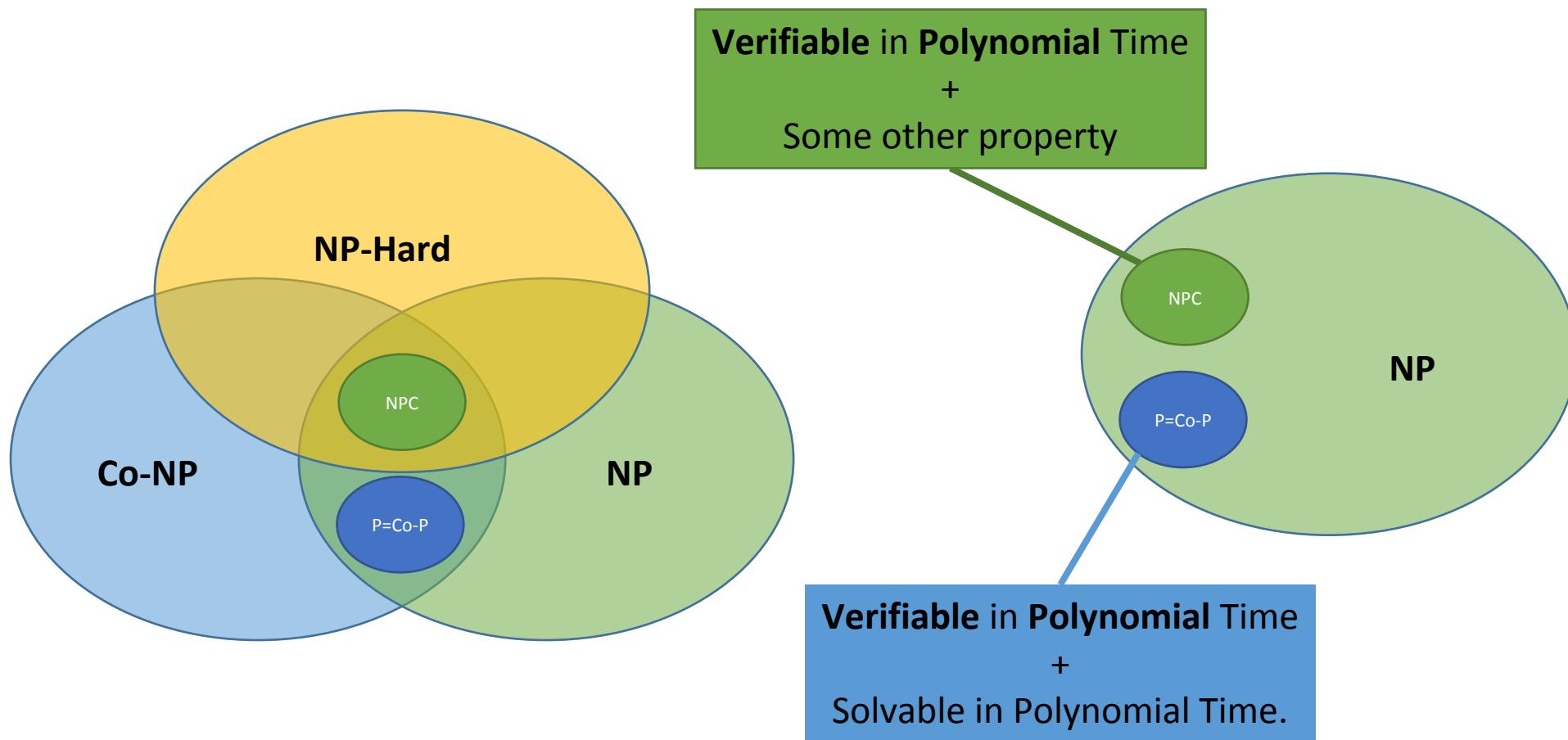
# NP Complete Problems (Cont.)

Let us take a look at the problems class Venn diagram again:



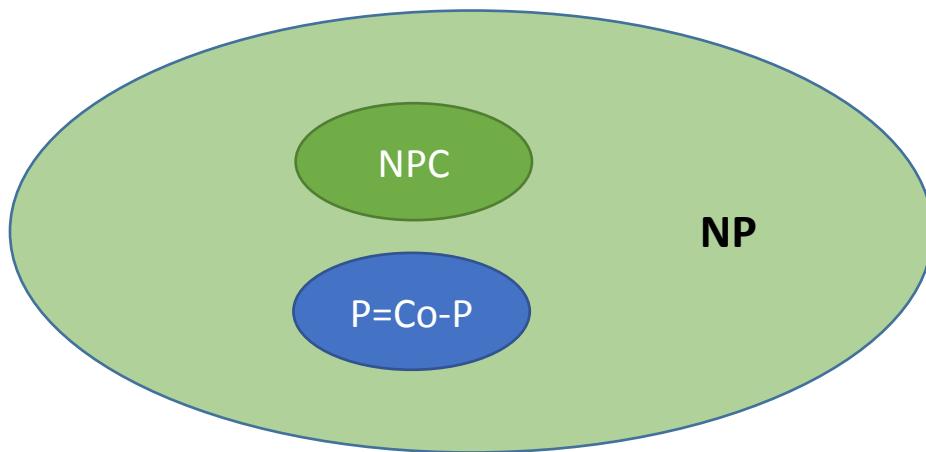
# NP Complete Problems (Cont.)

Let us take a look at the problems class Venn diagram again:



# NP Complete Problems (Cont.)

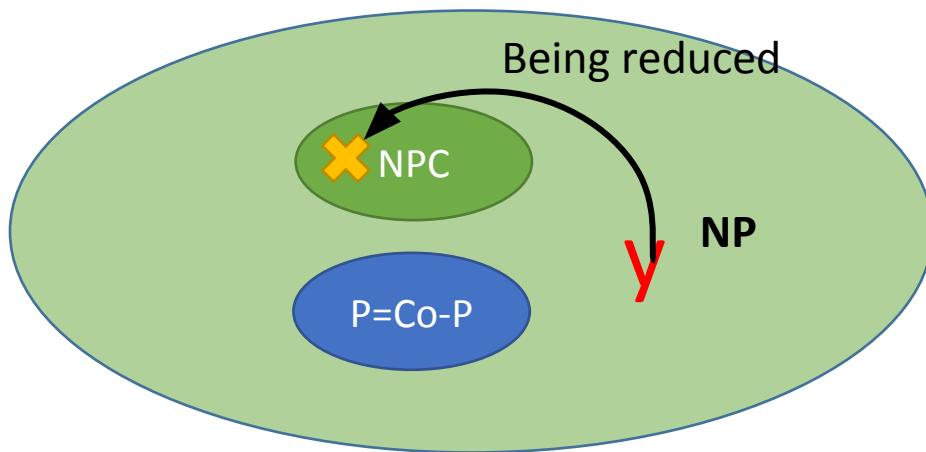
- What it is that **property**?



- Any **problem Y** in **NP class** can be reduced to any problem **X** in **NP complete** class.
- This means that the **inputs** of Y can be **changed** and made suitable for **input in problem X**. And then be solved.
- This **conversion** of inputs is done **in polynomial time**.

# NP Complete Problems (Cont.)

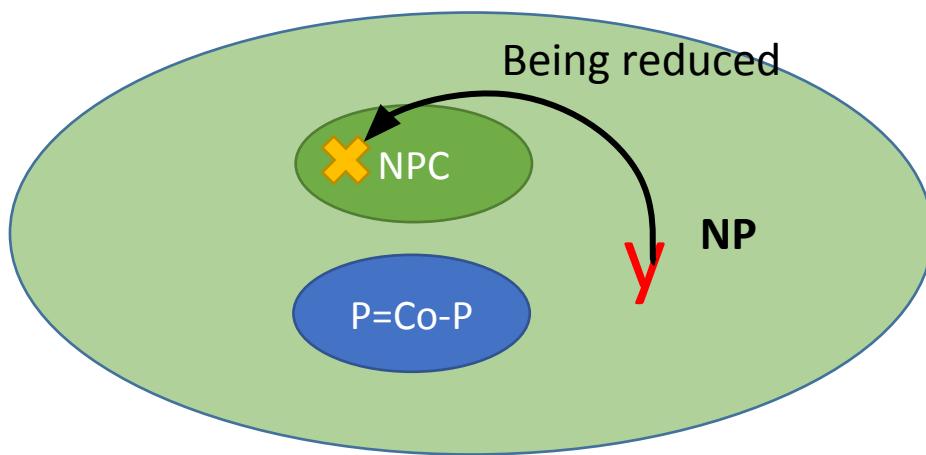
- What it is that **property**?



- Any problem **Y** in **NP** class can be reduced to any problem **X** in **NP complete** class.
- This means that the **inputs** of Y can be **changed** and made suitable for **input in problem X**. And then be solved.
- This **conversion** of inputs is done **in polynomial time**.

# NP Complete Problems (Cont.)

- What it is that **property**?



- Any problem **Y** in **NP** class can be reduced to any problem **X** in **NP complete** class.
- This means that the **inputs** of **Y** can be **changed** and made suitable for **input in problem X**. And then be solved.
- This **conversion** of inputs is done **in polynomial time**.

Note that if **Y** can be reduced to **X**, it **does not necessarily imply** that **X** can be reduced to **Y**.

# NP Complete Problems (Cont.)

The 3-SAT problem is an NP-complete problem.

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

For which values of  $x_i$  is it possible to make the entire expression equal to '1' ?

This is in NP class:

- There is **no polynomial time algorithm** for this.
- However, if someone gives us a solution, we can **verify** it in **polynomial time**.

But this is also NP complete:

- We can reduce **other problems** in **NP** into this problem.

# NP Complete Problems (Cont.)

The 3-SAT problem is an NP-complete problem.

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

Stephen Cook – Cook's Theorem.

- His **theorem proved that 3-SAT problem is an NP complete problem**
- It means any problem in NP class can be reduced to 3-SAT.
- 3-SAT – 3 Conjunctive Normal Form Satisfiability problem

# NP Complete Problems (Cont.)

## Reduction

The Super Mario Brothers game can be reduced to 3 SAT as well ([link](#)).

However, let us choose another problem to understand the intuition behind reduction.

# NP Complete (Cont.)

## Reduction (Intuition)

A problem in the NP set. Subset **sum problem**.

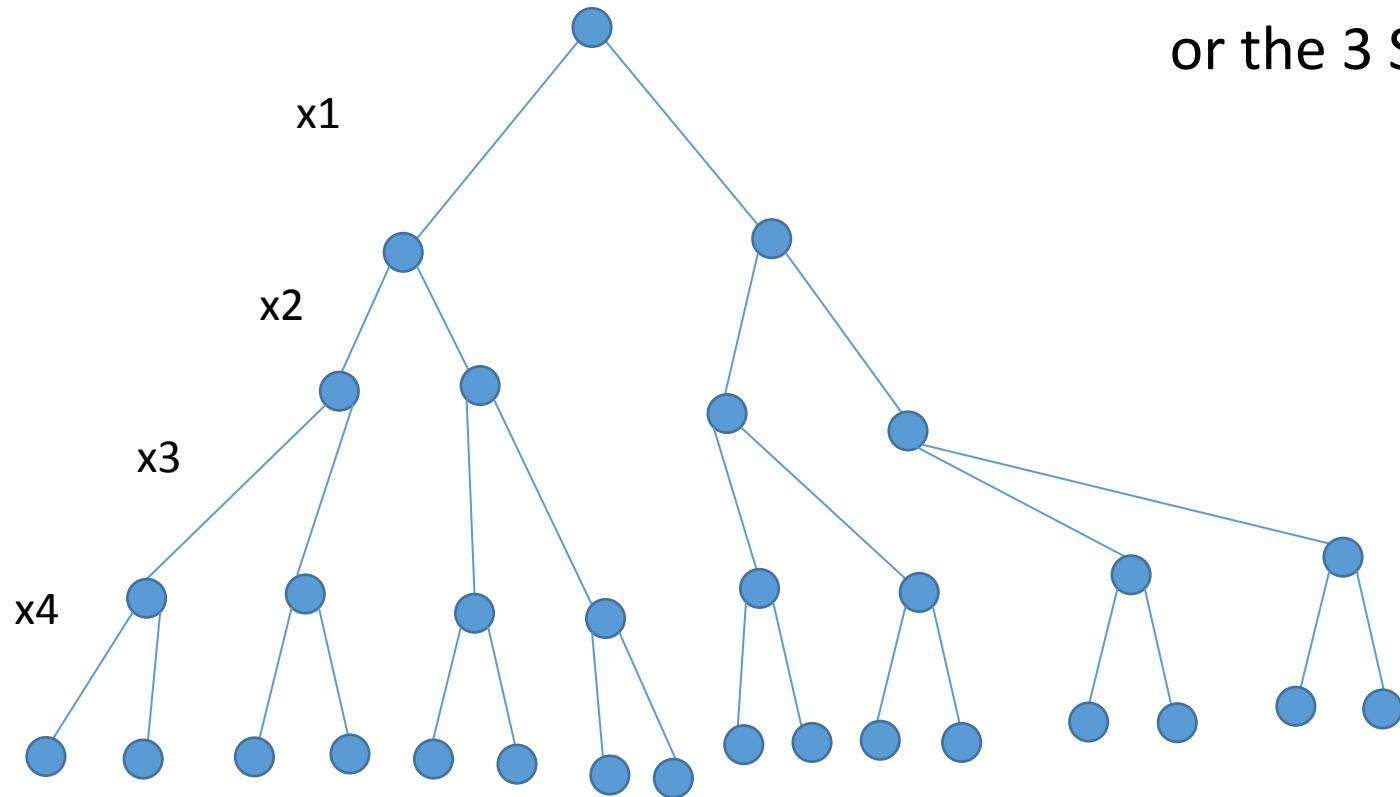
We want to check if some particular elements of the subset add up to some sum  $x$

# NP Complete (Cont.)

- Subset sum problem:
  - $S = \{3, 5, 10, 2, 1, 21\}$ ,  $n = 7$
  - Solution :  $X = \{0, 1, 0, 1, 0, 0\}$  – can be obtained by brute force – not in polynomial time
  - If a certificate (solution) is given we can verify it in polynomial time
- SAT problem:
  - $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$
  - Solution :  $\{1,1,0,0,1,0\}$  – can be obtained by brute force – not in polynomial time
  - If a certificate (solution) is given we can verify it in polynomial time

# NP Complete (Cont.)

## Reduction (Intuition)

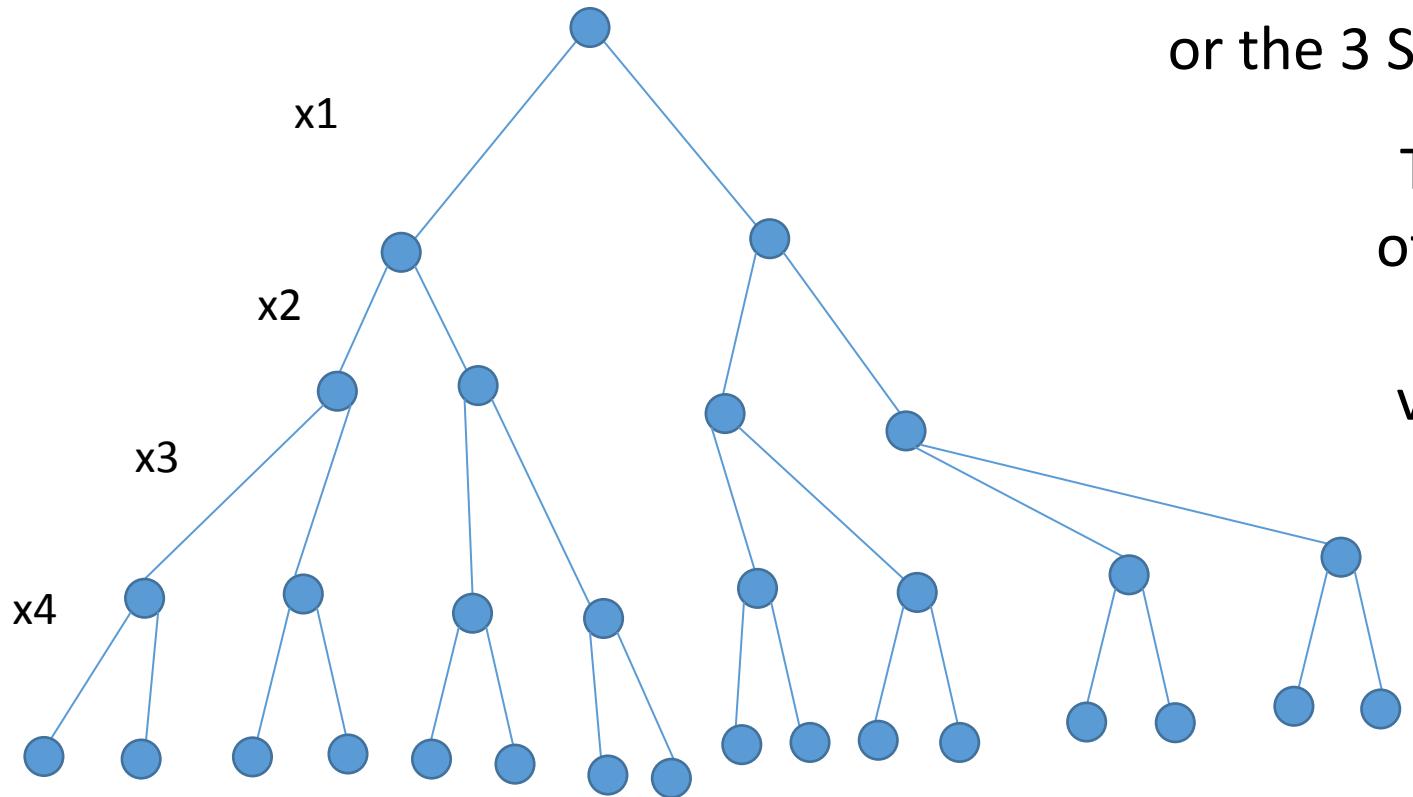


This can be the **brute force** exploration tree

For either subset sum problem  
or the 3 SAT problem

# NP Complete (Cont.)

## Reduction (Intuition)



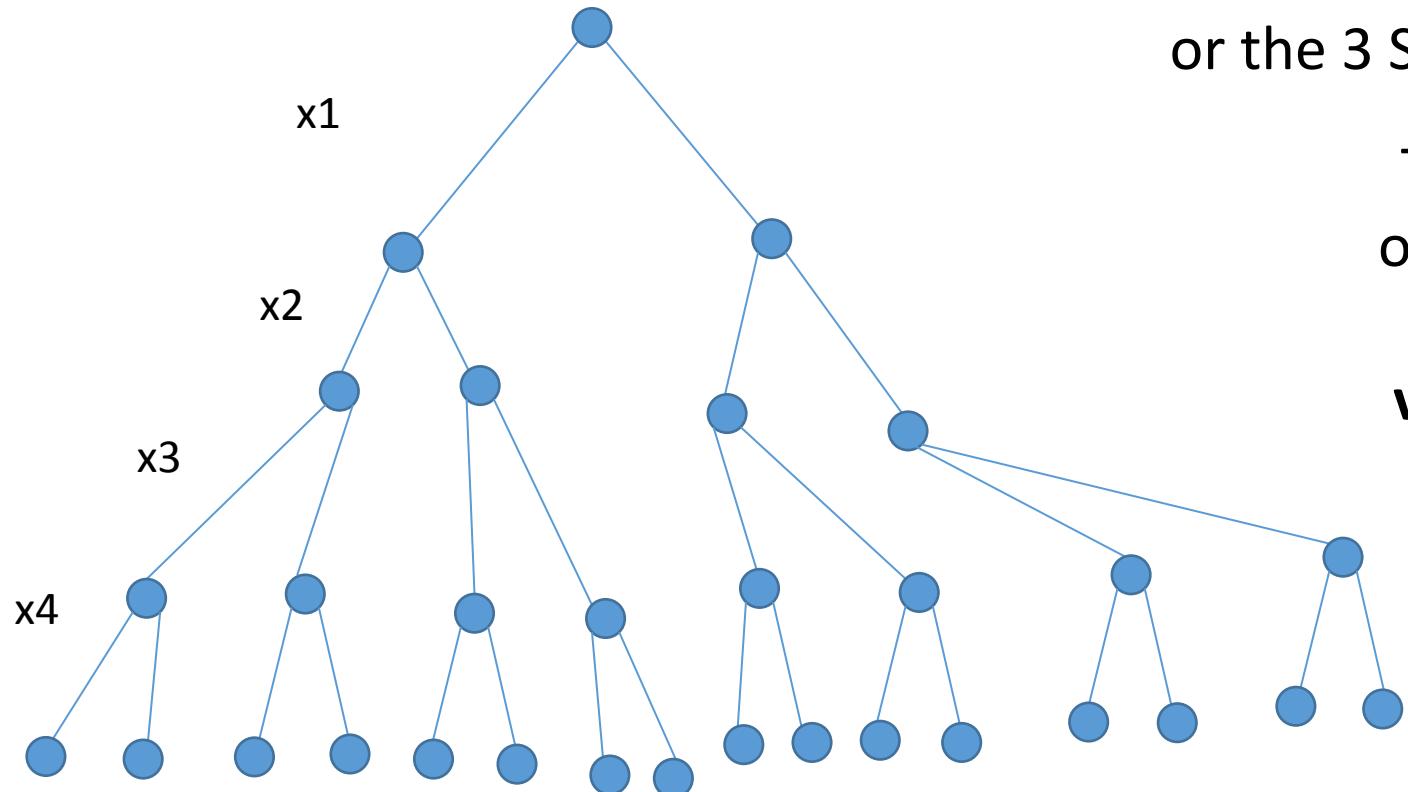
This can be the **brute force**  
exploration tree

For either subset sum problem  
or the 3 SAT problem

Thus we can reduce, the sum of subsets problem to NP 3-Sat problem. This is because the variables and steps are same.

# NP Complete (Cont.)

## Reduction (Intuition)



This can be the **brute force** exploration tree

For either subset sum problem  
or the 3 SAT problem

Thus we can **reduce**, the sum of subsets problem to NP 3-Sat problem. This is because the **variables and steps** are same.

Only the meaning of  $x_i$  has changed (it can mean the '0' or '1' value in case of 3 SAT or its element **taken/not taken** for sum of subsets problem)

# Why are NP complete problems important?

- Because, this chapter's name is np completeness.

# Why are NP complete problems important?

- This chapter's name is np completeness.
- All problems in **NP** set can be reduced to the problems in **NP-complete** set.
- Moreover, a problem in **NP-complete** set can be reduced to another problem in the **NP-complete** set. (as NP-complete class problems are NP class problems as well.)
- If a problem in **NP-complete** set can be solved in **polynomial** time.
- Then all **problems in NP** can be solved in polynomial time.
- Then P will no longer be a subset of NP. **NP=P** will be true.

# Useful Link

[https://www.cs.lmu.edu/cs4321/www/Lectures/Lecture%2025%20-%20P%20and%20NP.htm?fbclid=IwAR2IoKQdaNhdJLGrOstsVjtXQwVNP4DsNb\\_e4XS-qErfLOvMMDWx2nyKIL8](https://www.cs.lmu.edu/cs4321/www/Lectures/Lecture%2025%20-%20P%20and%20NP.htm?fbclid=IwAR2IoKQdaNhdJLGrOstsVjtXQwVNP4DsNb_e4XS-qErfLOvMMDWx2nyKIL8)

# 4. NP-Hard, Co-P and Co-NP classes.

# NP Hard problem

- These are the problems that are **at least as hard** as NP complete problems. But can be **harder**. (we will understand this statement a bit later)
- NP and NP complete problems can be attempted to be solved using different **strategies**.
- The strategies **may yield a solution** which will can be **verified** in polynomial time. (example : graph coloring)
- But for NP hard problems this is **not always the case (soln verification)**.
- **NP complete** problems can be **reduced to NP – hard** problems

# NP Hard problem

Summary:

- **NP complete** problems can be **reduced to NP – hard** problems.
- Cannot be solved in polynomial time.
- Solutions may or may not be verified.

# NP Hard problem

NP hard problem example:

**The halting problem:**

- The [halting problem](#) is an NP-hard problem. This is the problem that given a **program P** and **input I**, will it **halt**? This is a decision problem but it is not in NP.
- No way to answer that question in polynomial time.
- No specific way to verify if the program halted, whether it really halted.
- Observe that these problems are a bit **abstract** in nature as well.

# Co-P

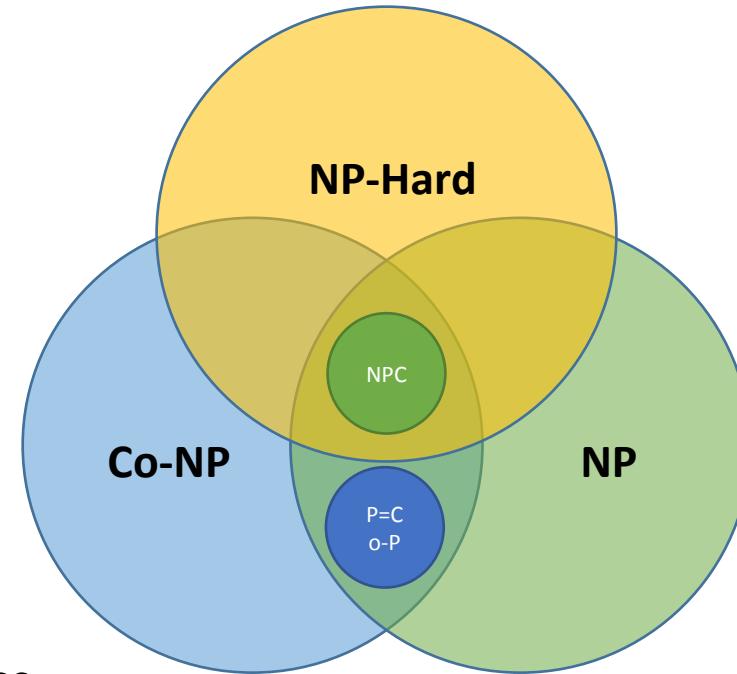
- Complement of P class problems.
- Share the same properties as P.
- Hence,  $P=Co-P$
- Sometimes not included in Venn Diagram

**Example 1 :** finding out the prime number in a range

**Example 2 :** finding out the non-prime numbers in a range

# Co-P

- Complement of P class problems.
- Share the same properties as P.
- Hence,  $P=Co-P$
- Sometimes not included in Venn Diagram

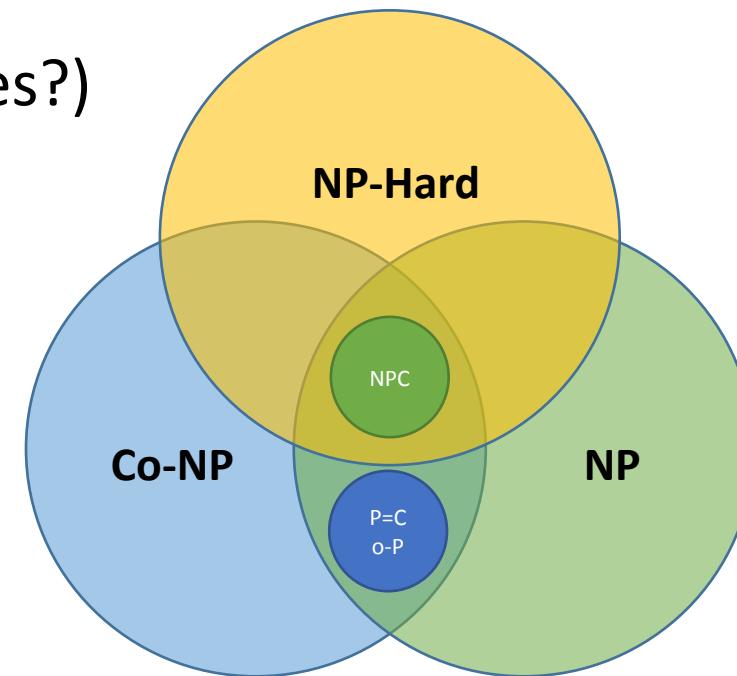


**Example 1 :** finding out the prime number in a range

**Example 2 :** finding out the non-prime numbers in a range

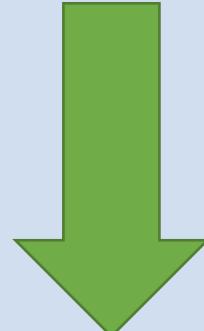
# Co-NP

- Complement of NP
- Is Co-NP = NP? (i.e. share same properties?)
- Not as per the diagram.
- But this is still a debatable question.
- **Co-P** is a subset of Co-NP.
- This class is not yet all defined.
- Sometimes **not included** in the Venn diagram



## 5. Associated Level of Difficulty and Coping Mechanisms

# Associated Difficulty Level of the Problem Classes

Problem Class	Increasing Difficulty
P	
NP	
NP-Complete	
NP-Hard	

# Coping strategies for increasing difficulties

## Brute-force algorithms

- Backtracking : Develop clever **enumeration** strategies
- **Guaranteed** to find optimal **solution**
- **No guarantees** on running time

## Heuristics (AI course)

- Distance from the **goal state**.
- **Not Guaranteed** to run in **polynomial** time
- **No guarantees** on quality of solution
- Example : 8 Puzzle Problem

## Approximation algorithms

- **Guaranteed** to run in **polynomial time**
- **Guaranteed** to find "high quality" solution, say within 10% of optimum
- **Obstacle:** need to prove a solution's value is **close to optimum**, without even knowing what optimum value is !

# 5 Examples of NP Complete Problems:

1. Circuit Satisfiability problem 3 SAT. (**covered**)
2. Hamiltonian Cycle Problem. (**covered**)
3. Travelling Salesman Problem
4. 3-Coloring Problem (**covered**)
5. Longest Path Problem (**issues with cycles**)

# 5 Examples of NP Complete Problems:

1. Circuit Satisfiability problem 3 SAT. (**covered**)
2. Hamiltonian Cycle Problem. (**covered**)
3. Travelling Salesman Problem
4. 3-Coloring Problem (**covered**)
5. Longest Path Problem (**issues with cycles**)

SINCE THEY ARE NP COMPLETE class problems, they are also NP and NP Hard class problems.

# 6. Approximation Problems

# Approximation Algorithm

- **Our Goal** : Solve an optimization problem in such a way that even if the optimal value is not achieved, a **near optimal** value can be obtained.
- A term – approximation ratio
  - $C^*$  - suppose this is the optimal value.
  - $C$  – suppose this is the near optimal value.
  - Approximation ratio – denoted by the variable  $\rho$

# Approximation Algorithm

A term – approximation ratio

- $C^*$  - suppose this is the optimal value.
- $C$  – suppose this is the near optimal value.
- Approximation ratio – denoted by the variable  $\rho$

Note that the relation between  $C$  and  $C^*$  varies based on optimization type:

- $C^*$  is greater than  $C$  if the problem is a maximization problem
- $C^*$  is less than  $C$  if the problem is a minimization problem

# Approximation Algorithm

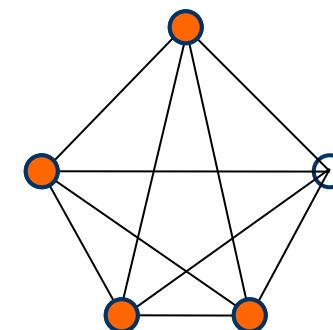
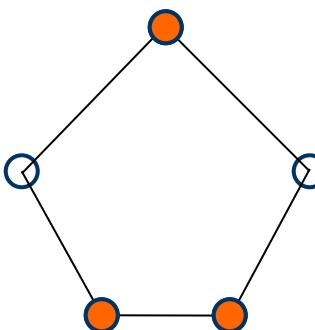
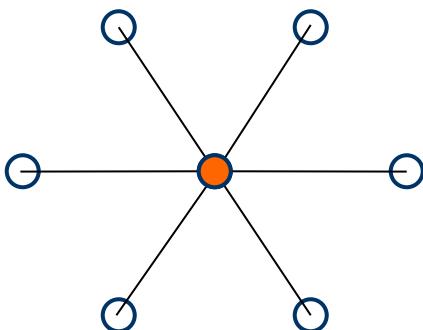
- An approximate algorithm has an **approximation ratio** of  $\rho$  if:  
$$\max\{C/C^*, C^*/C\} \leq \rho$$
- The inequality is like this because it takes care of both maximization and minimization problems.
- Intuition: Suppose you have developed another 0/1 knapsack algorithm. For some particular input, the max profit attainable is 100 dollars. However, your algorithm is based on approximation and is able to generate maximum profit of 50 (may generate even less profit).
- Then by the above relation, your approximation algorithm has an approximation ratio of 2.

# Approximation Algorithm

**Vertex-Cover:** a subset of vertices which “**covers**” every edge.  
An edge is **covered** if one of its endpoints is chosen.

## The Vertex-Cover Problem:

Find a vertex cover with **minimum** number of vertices.



# Approximation Algorithm

## What is a vertex-cover ?

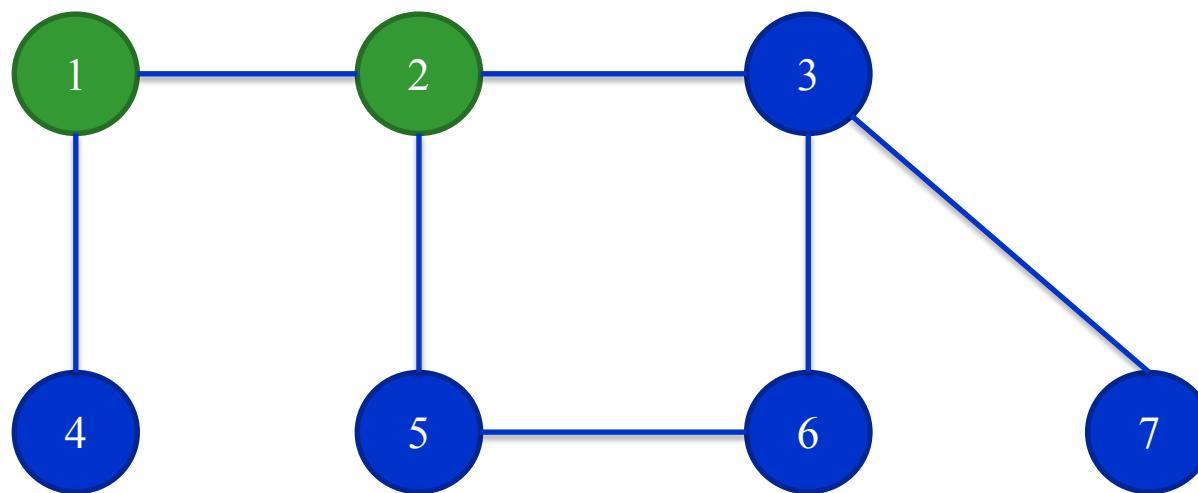
- Given an undirected graph  $G = (V, E)$ , a vertex-cover  $V'$
- $V' \subseteq V$
- For each edge  $(u, v)$  in  $E$ , either  $u \in V'$  or  $v \in V'$  or both

## What is the Vertex-Cover Problem ?

- Find a vertex cover with minimum number of vertices

# Approximation Algorithm

- Trying to solve this using approximation algorithm



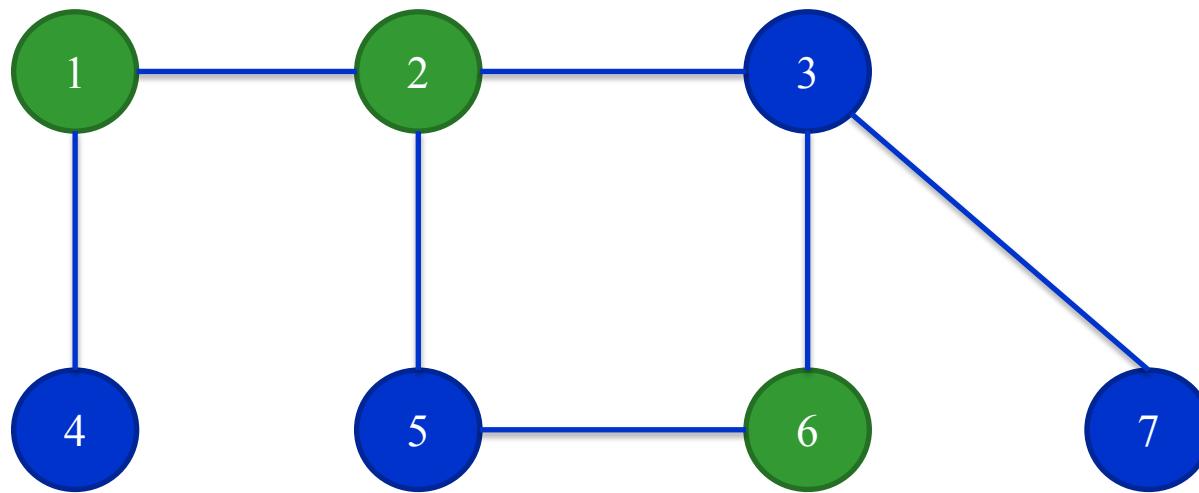
Are the green vertices a vertex-cover?

No. why?

Edges (5, 6), (3, 6) and (3, 7) are not covered by it

# Approximation Algorithm

- Trying to solve this using approximation algorithm



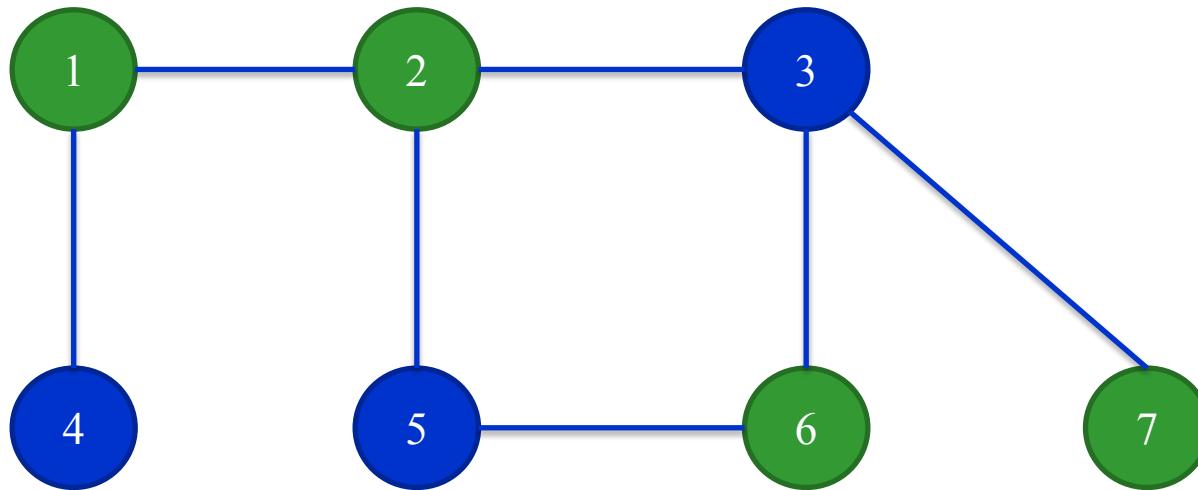
Are the green vertices a vertex-cover?

No. why?

Edge (3, 7) is not covered by it

# Approximation Algorithm

- Trying to solve this using approximation algorithm

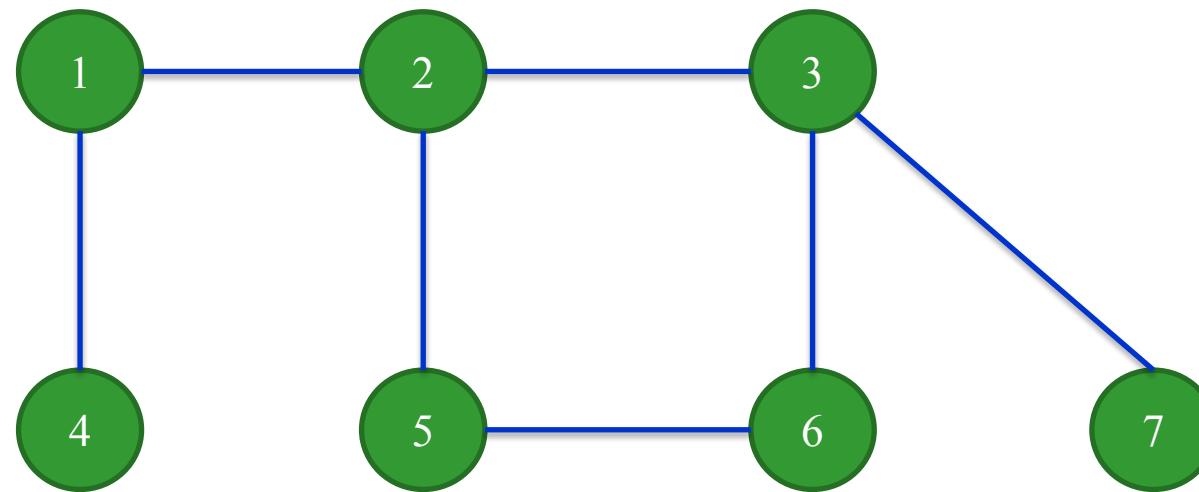


Are the green vertices a vertex-cover? Yes

What is the size? 4

# Approximation Algorithm

- Trying to solve this using approximation algorithm



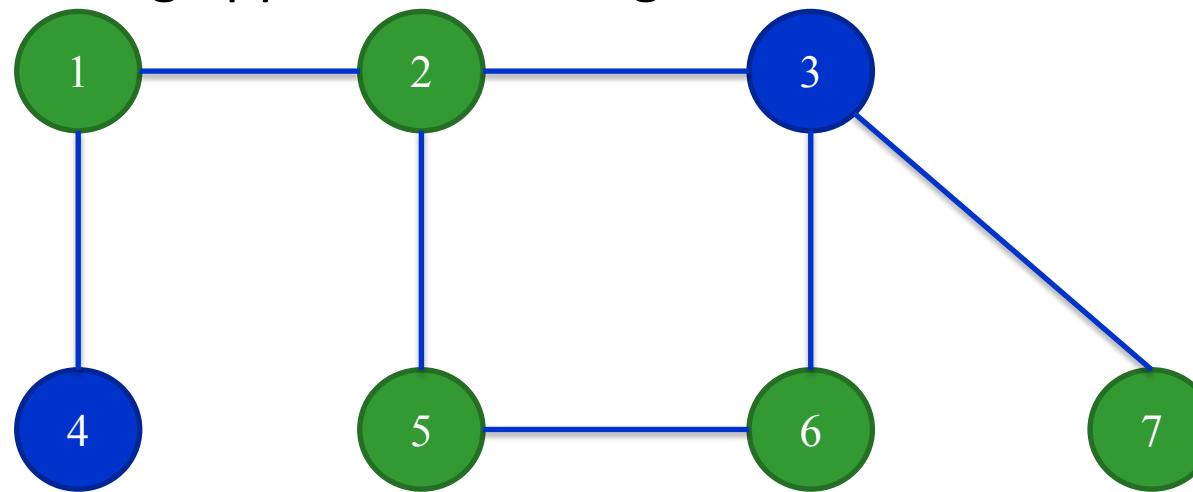
Are the green vertices a vertex-cover?

Yes

What is the size? 7

# Approximation Algorithm

- Trying to solve this using approximation algorithm



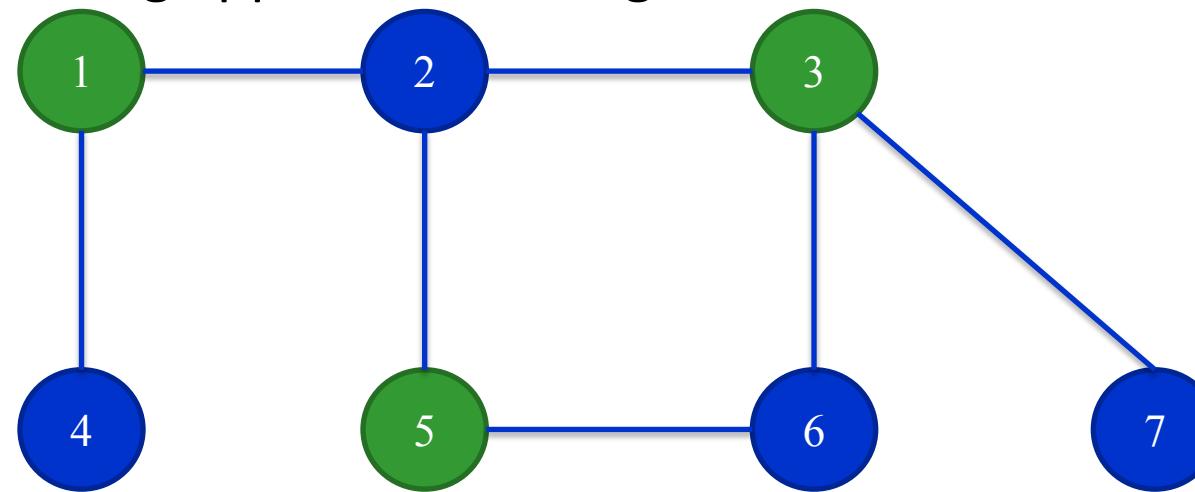
Are the green vertices a vertex-cover?

Yes

What is the size? 5

# Approximation Algorithm

- Trying to solve this using approximation algorithm



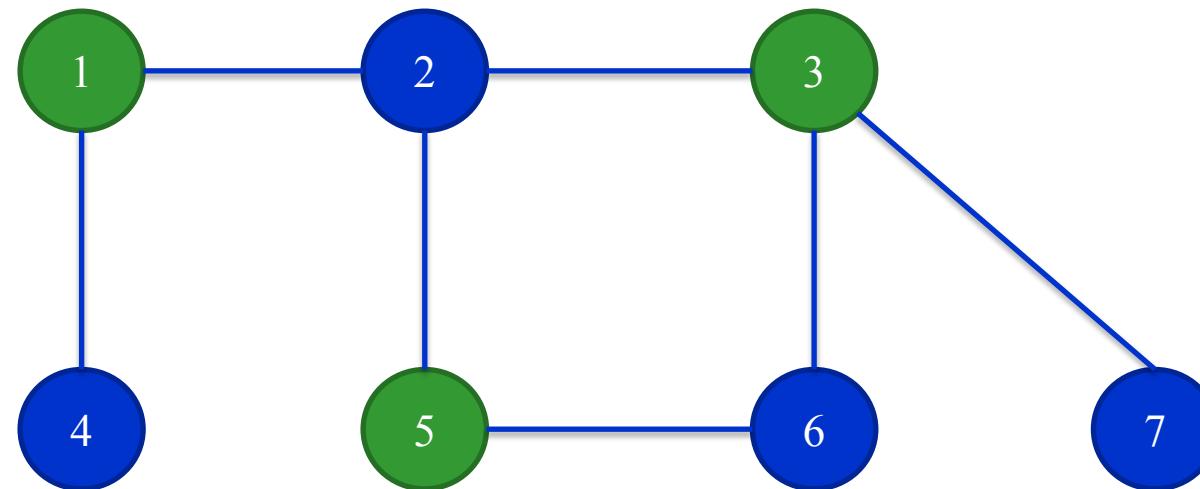
Are the green vertices a vertex-cover?

Yes

What is the size? 3

# Approximation Algorithm

- Trying to solve this using approximation algorithm



A minimum vertex-cover

THE OPTIMAL SOLUTION :3

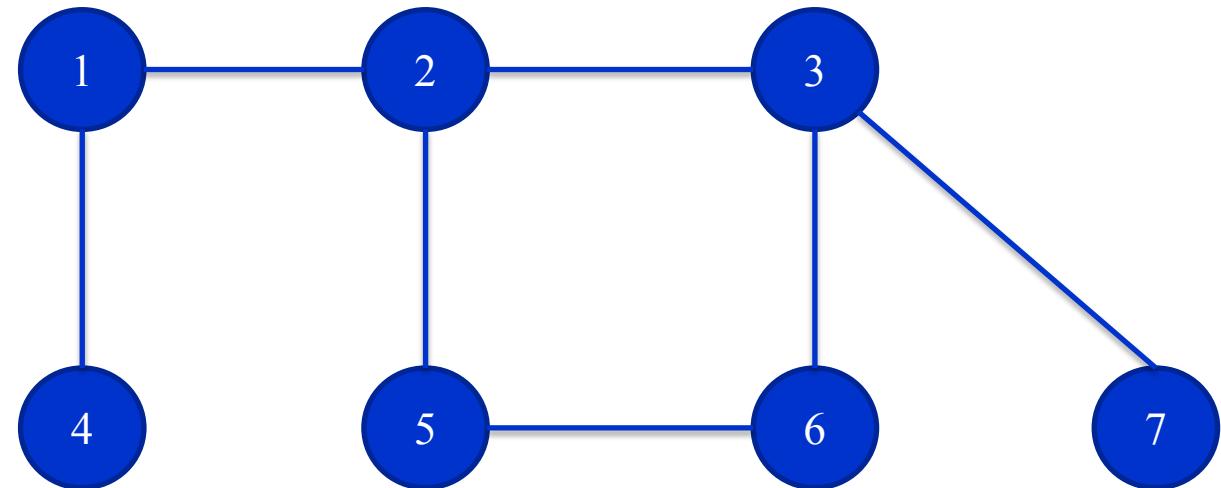
# The Algorithm Pseudocode

```
APPROX-VERTEX-COVER(G)
    C = Ø;
    E' = G.E;
    while(E' ≠ Ø ){
        Randomly choose a edge
        (u,v) in E', put u and v
        into C;
        Remove all the edges
        that covered by u or v
        from E'
    }
    Return C;
```

# The Algorithm Pseudocode

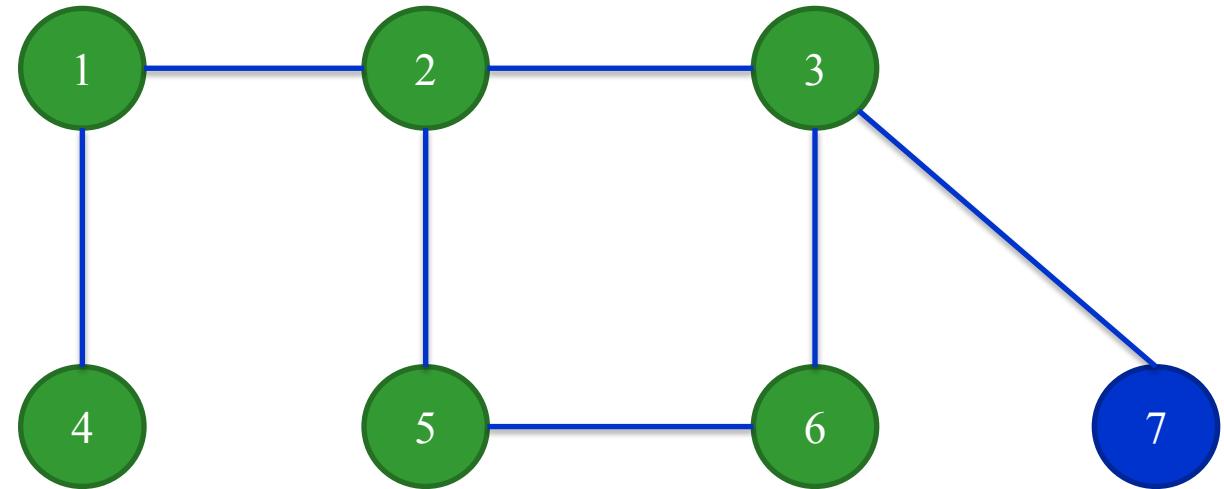
**APPROX-VERTEX-COVER(G)**

```
C = Ø;  
E' = G.E;  
while(E' ≠ Ø ){  
    Randomly choose a edge  
    (u,v) in E', put u and v  
    into C;  
    Remove all the edges  
    that covered by u or v  
    from E'  
}  
Return C;
```



# The Algorithm Pseudocode

```
APPROX-VERTEX-COVER(G)
    C =  $\emptyset$ ;
    E' = G.E;
    while( $E' \neq \emptyset$ ){
        Randomly choose a edge
         $(u,v)$  in  $E'$ , put  $u$  and  $v$ 
        into  $C$ ;
        Remove all the edges
        that covered by  $u$  or  $v$ 
        from  $E'$ 
    }
    Return  $C$ ;
```



It is then a vertex cover

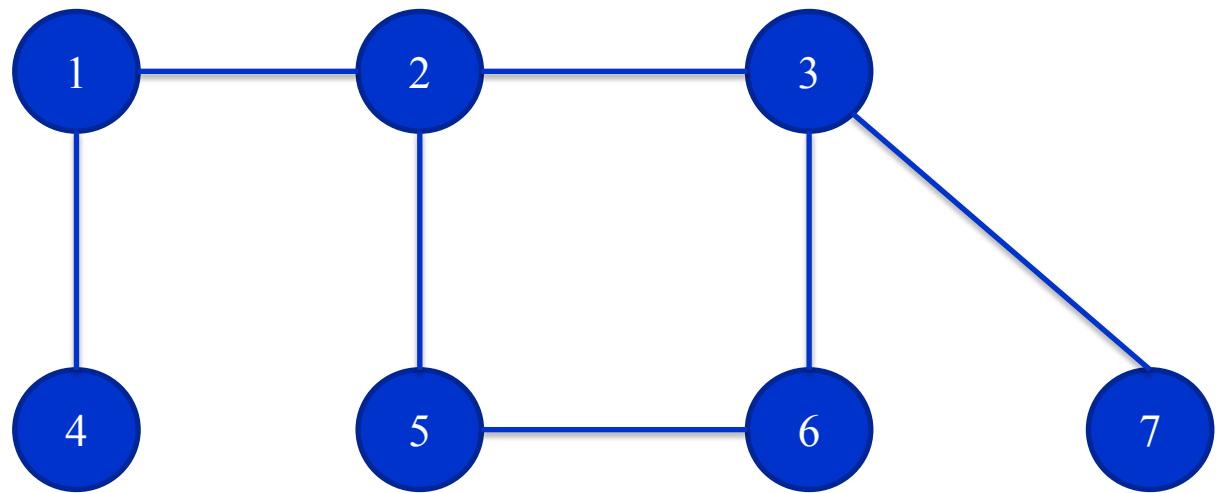
Size? 6

How far from optimal one?  $\text{Max}(6/3, 3/6) = 2$

# The Algorithm Pseudocode

**APPROX-VERTEX-COVER(G)**

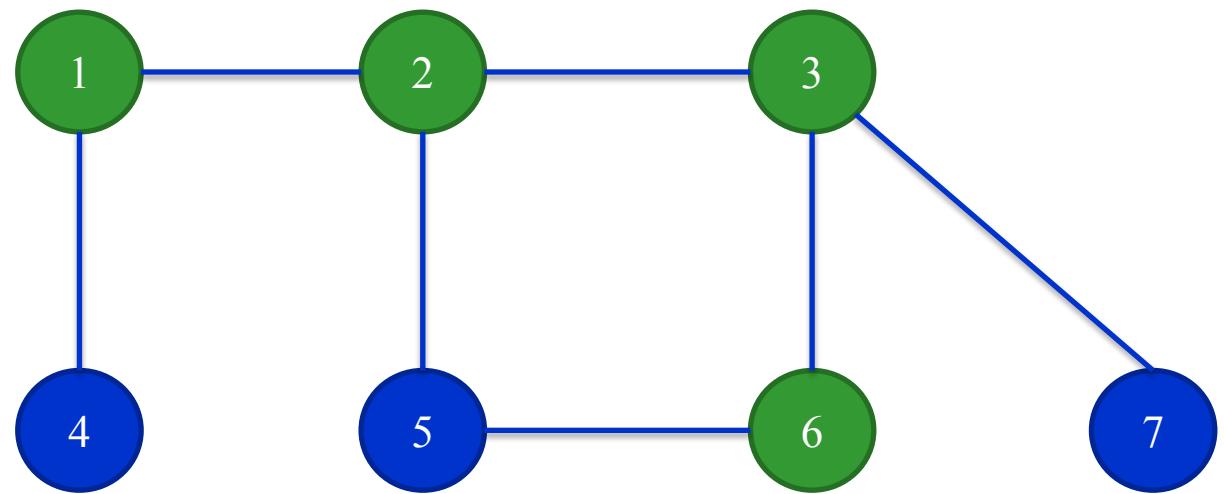
```
C = Ø;  
E' = G.E;  
while(E' ≠ Ø ){  
    Randomly choose a edge  
    (u,v) in E', put u and v  
    into C;  
    Remove all the edges  
    that covered by u or v  
    from E'  
}  
Return C;
```



# The Algorithm Pseudocode

**APPROX-VERTEX-COVER(G)**

```
C = Ø;  
E' = G.E;  
while(E' ≠ Ø ){  
    Randomly choose a edge  
    (u,v) in E', put u and v  
    into C;  
    Remove all the edges  
    that covered by u or v  
    from E'  
}  
Return C;
```



It is then a vertex cover

Size? 4

How far from optimal one?  $\text{Max}(4/3, 3/4) = 1.33$

## **APPROX-VERTEX-COVER(G)**

```
C = Ø;  
E' = G.E;  
while(E' ≠ Ø ){  
    Randomly choose a edge  
    (u,v) in E', put u and v  
    into C;  
    Remove all the edges  
    that covered by u or v  
    from E'  
}  
Return C;
```

### **Note:**

The edges that we selected are now in A.

The vertices attached to those edges are in C.

This observation will help in formulating an equation.

# Approximation Algorithm

**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

1.  $|C^*| \geq |A|$ 
  - A is the set of edges we selected during the algorithm
  - C is the number of vertex covers we found
  - $C^*$  is the optimal number of vertex covers we found

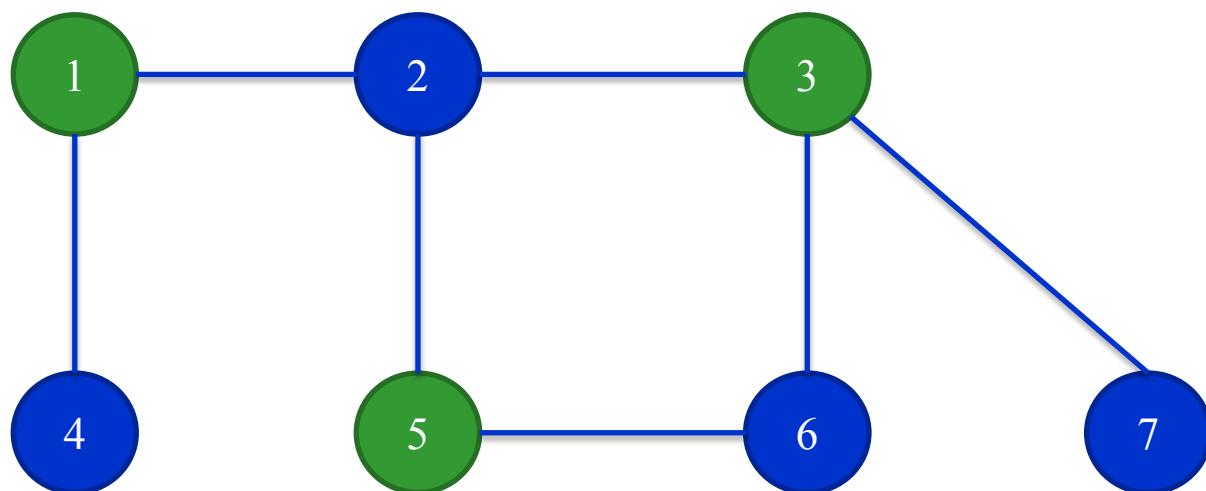
# Approximation Algorithm

**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$



# Approximation Algorithm

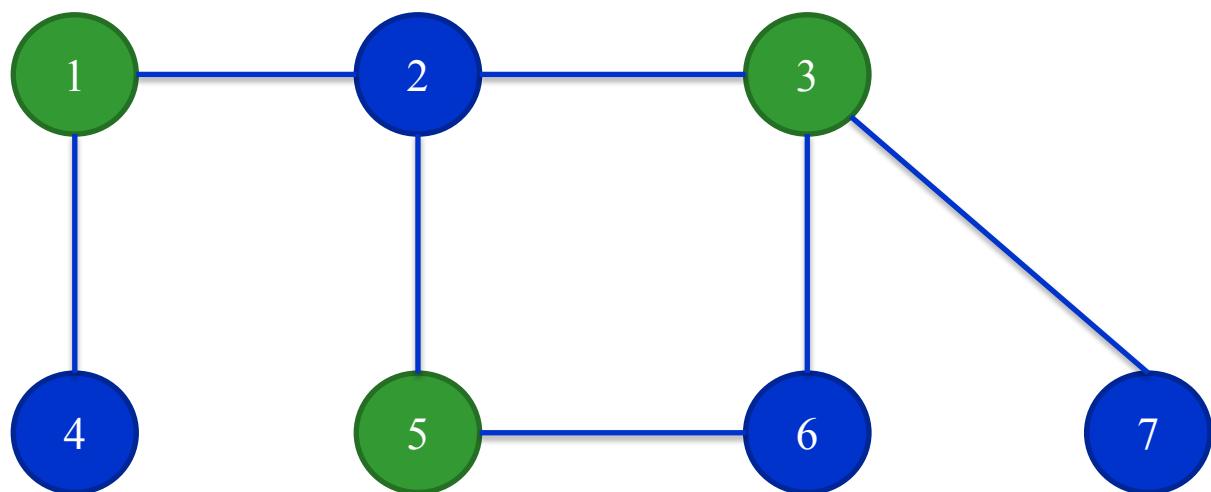
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

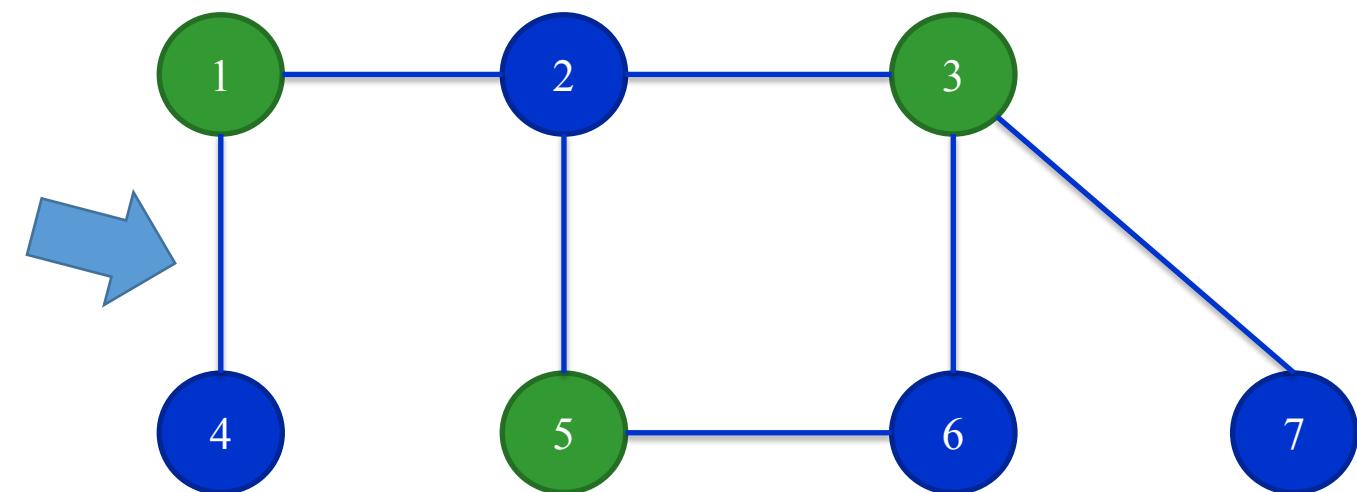
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

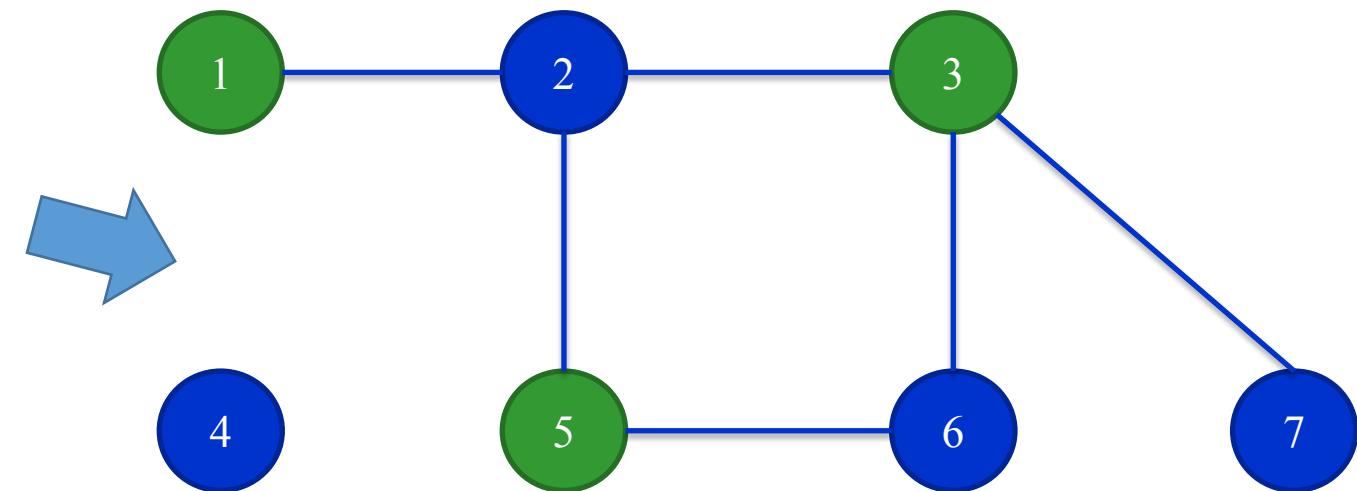
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

1.  $|C^*| \geq |A|$

$C^*=3$

Example for  $|C^*|=|A|$



# Approximation Algorithm

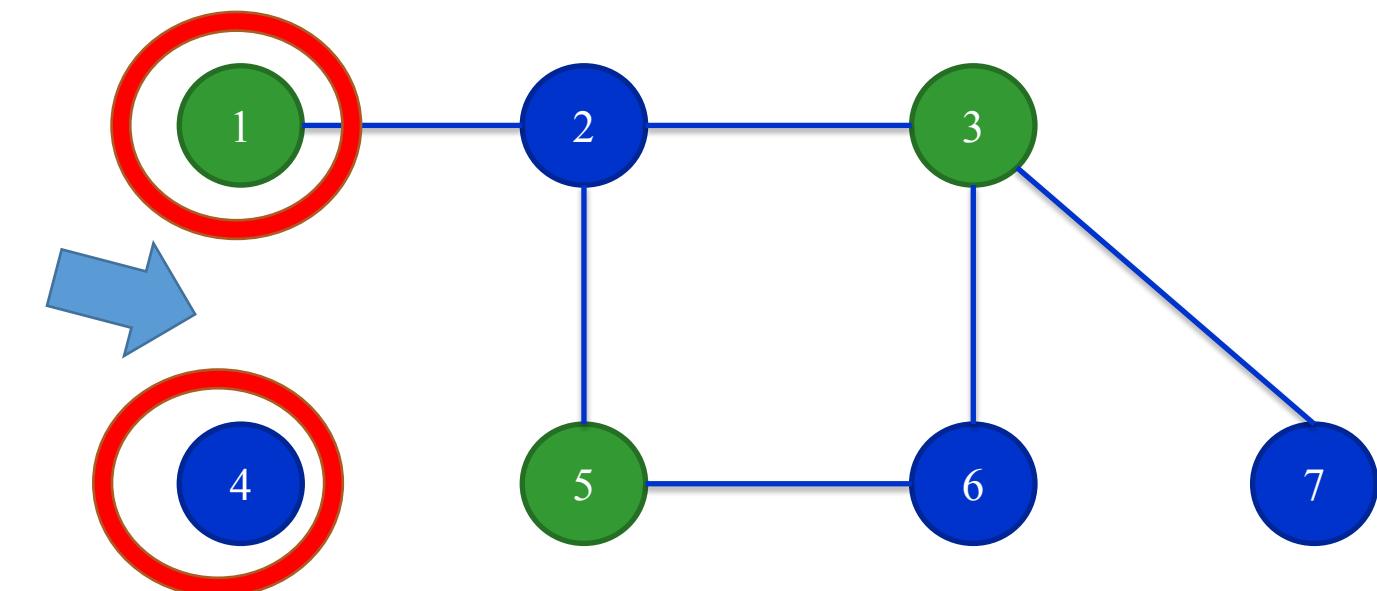
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

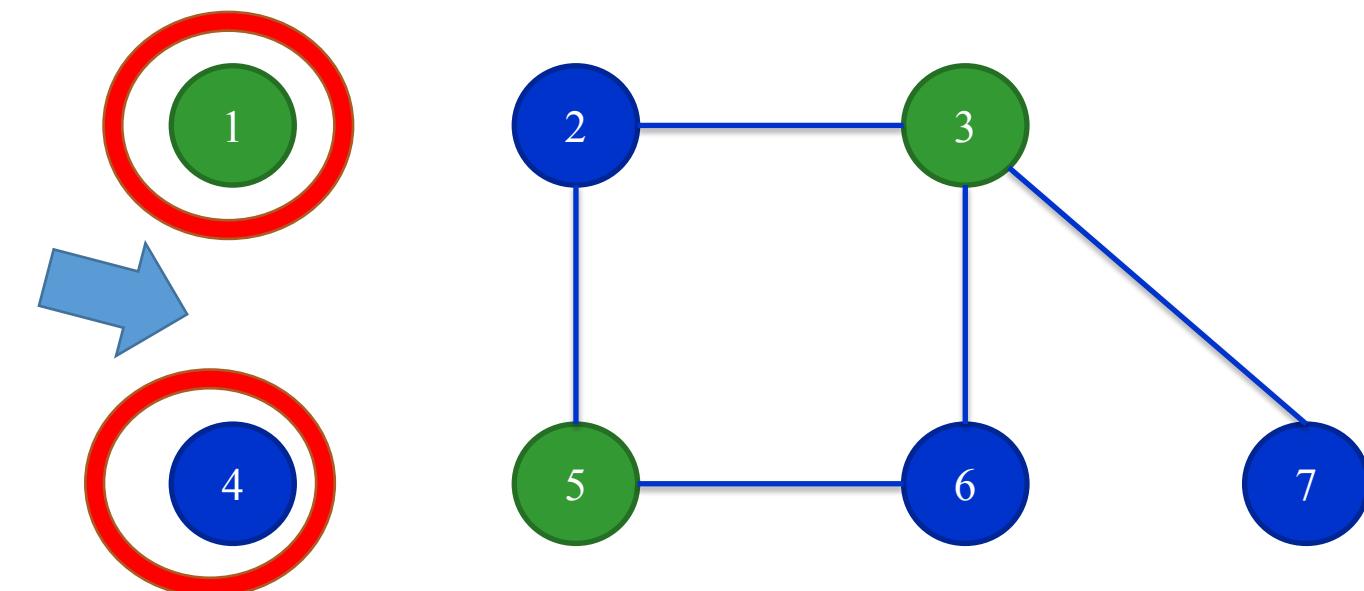
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

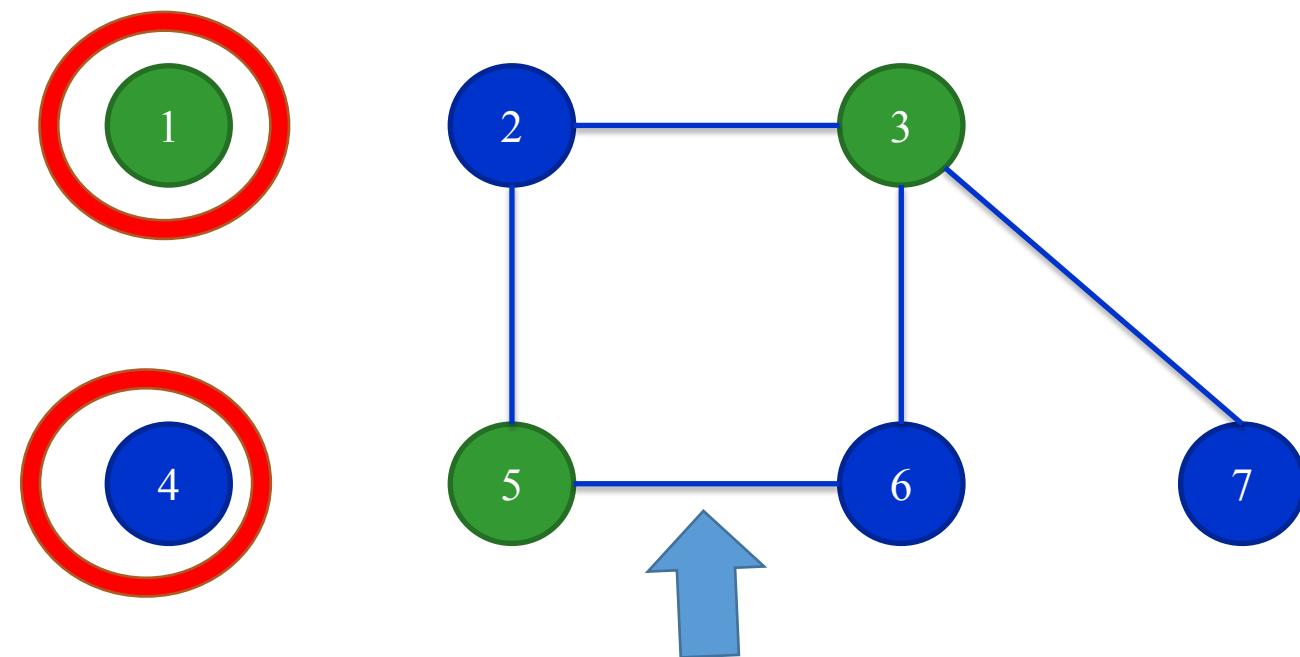
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

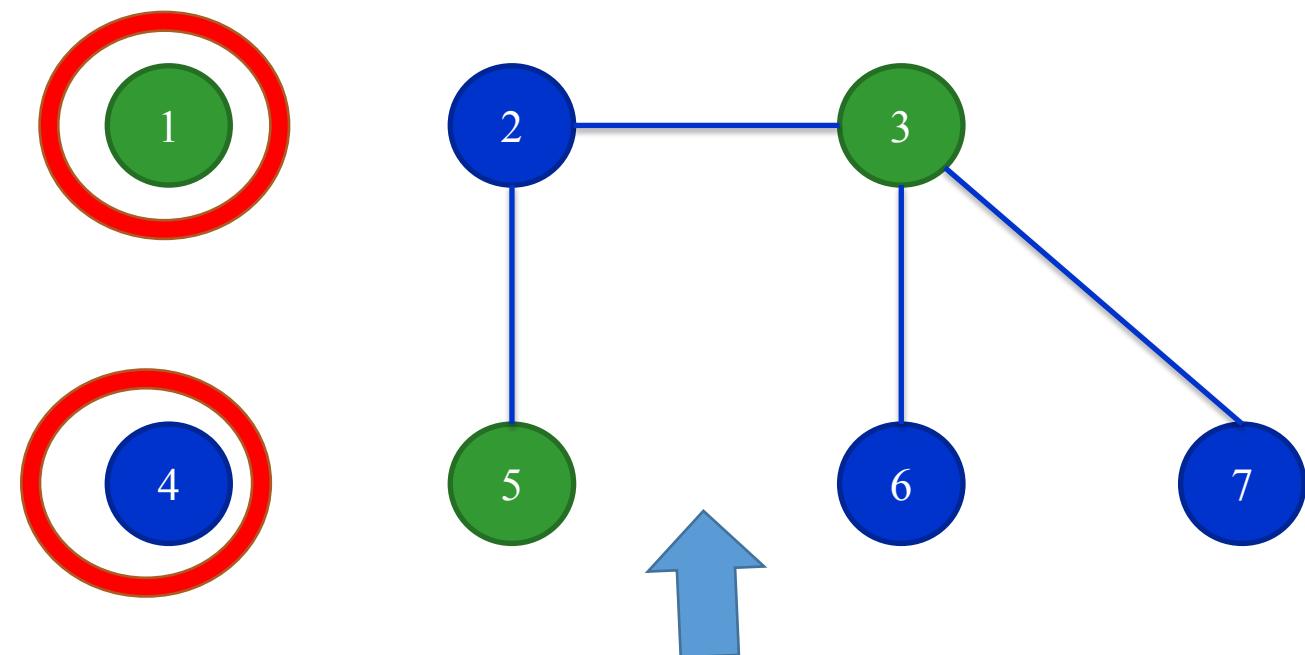
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

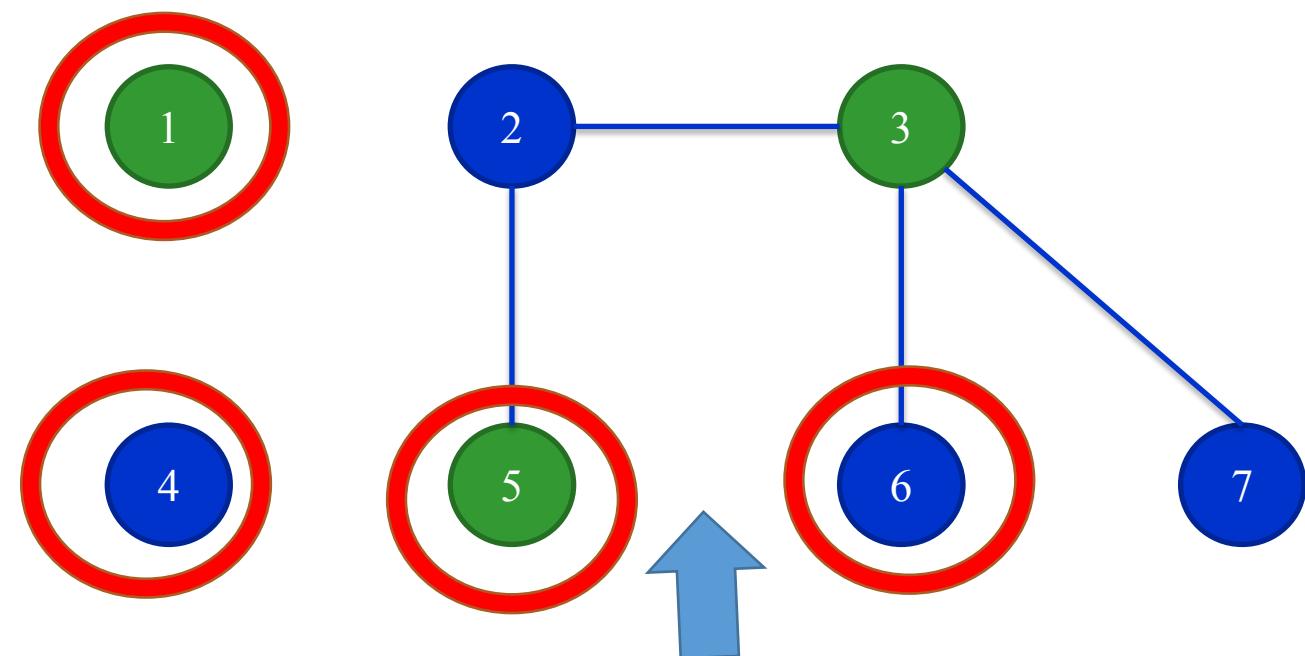
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

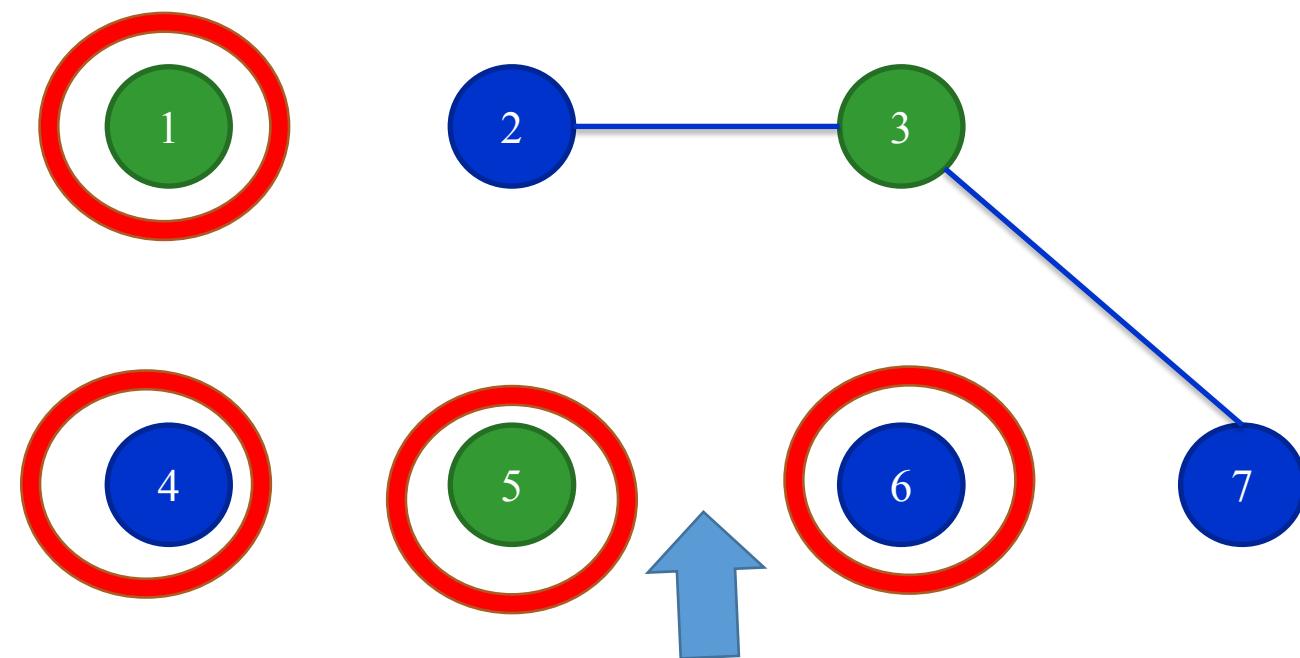
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

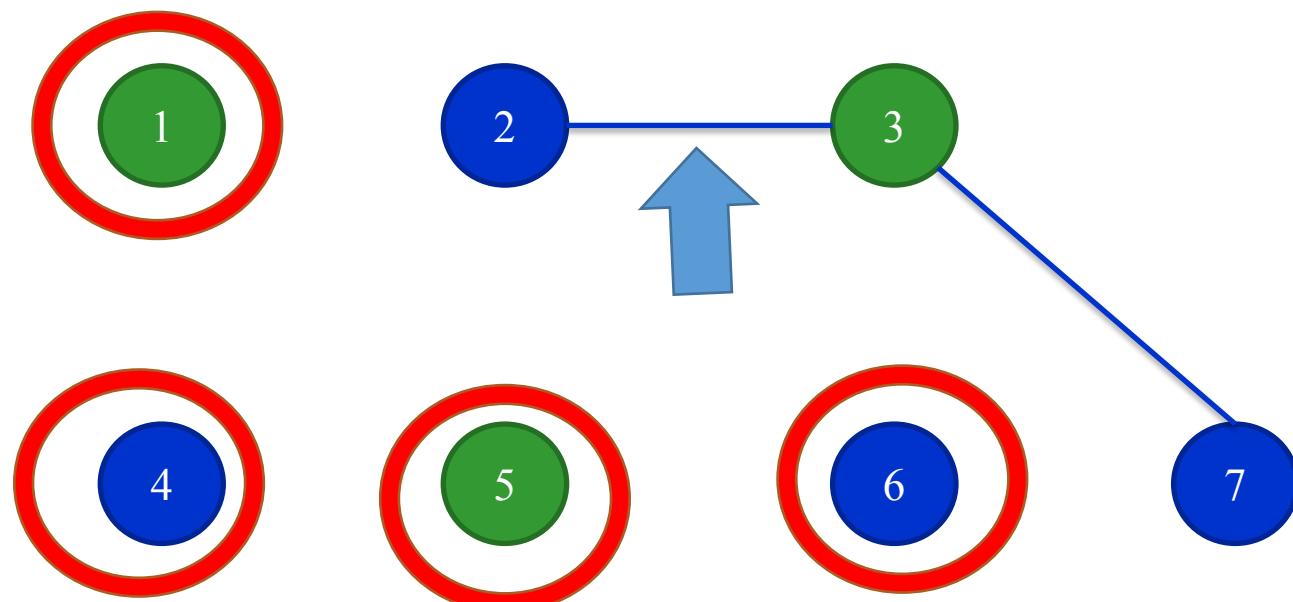
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

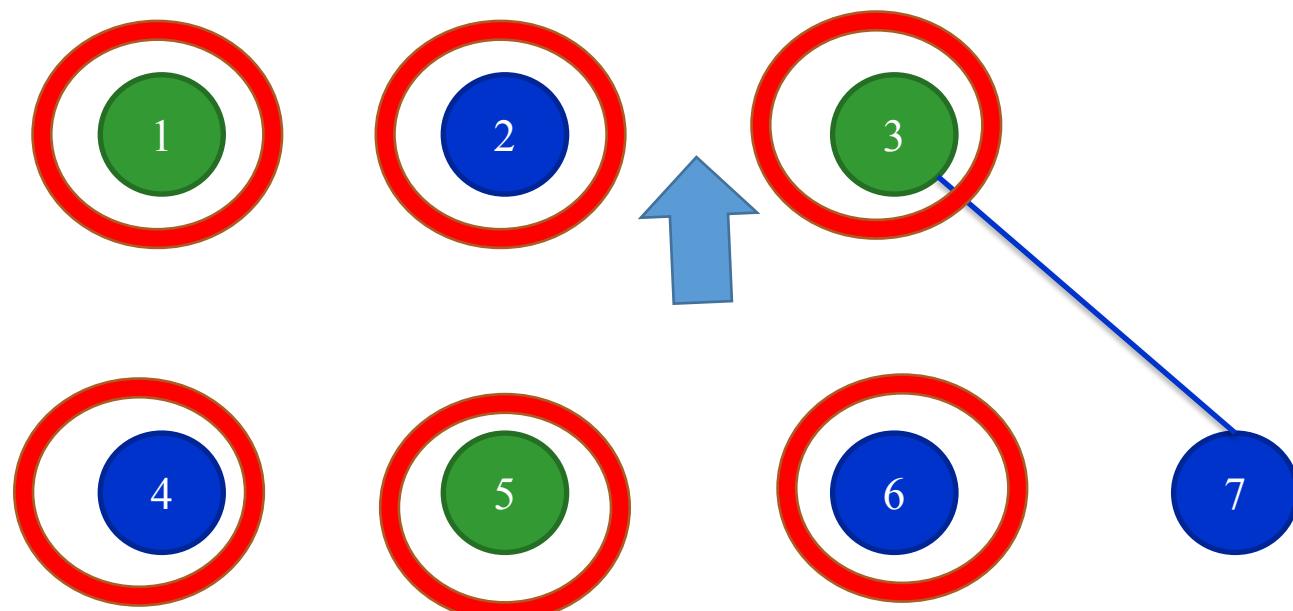
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

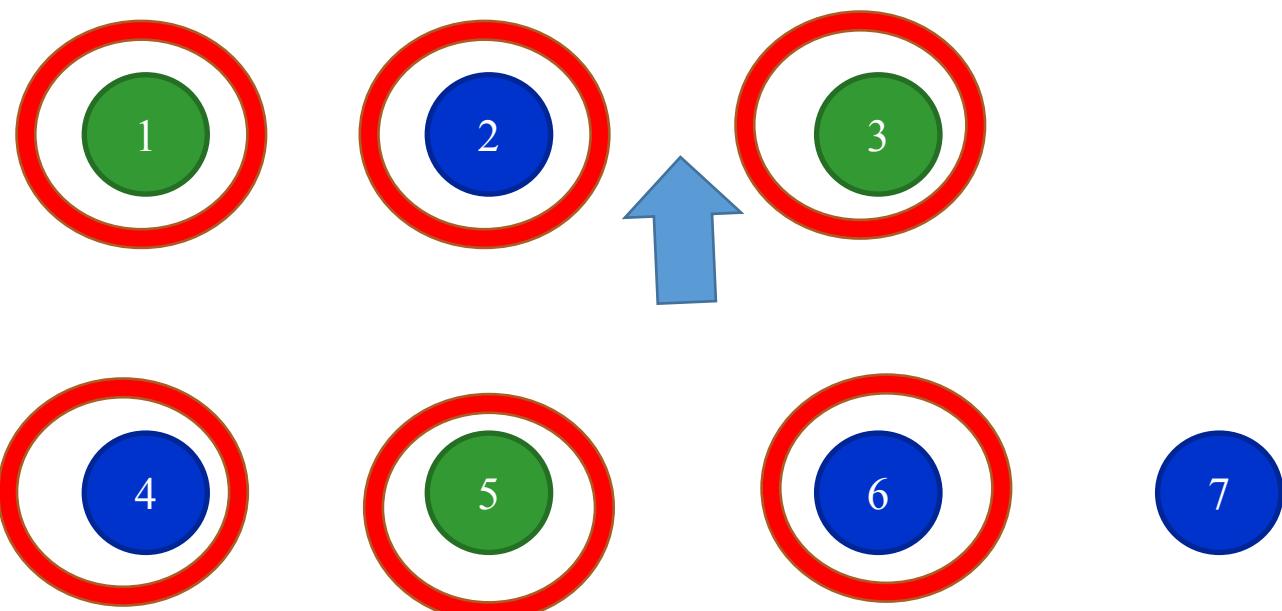
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*|=|A|$



# Approximation Algorithm

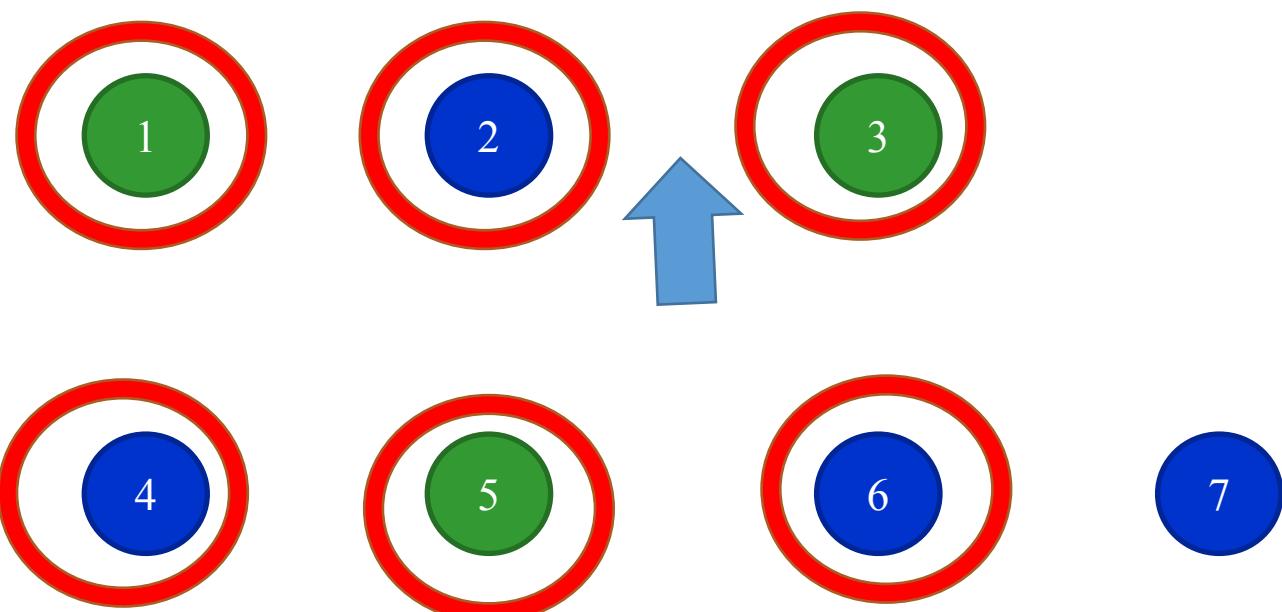
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

**Example for  $|C^*|=|A|$**



We selected 3 edges and managed to find 6 covers C including the 3 optimal ones  $C^*$

# Approximation Algorithm

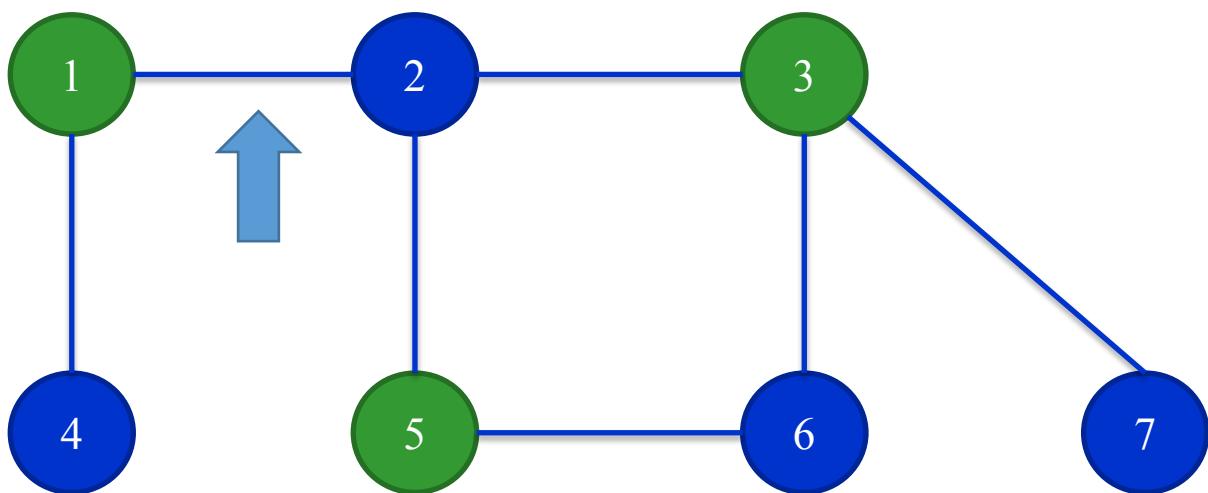
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

1.  $|C^*| \geq |A|$

$C^*=3$

Example for  $|C^*| > |A|$



# Approximation Algorithm

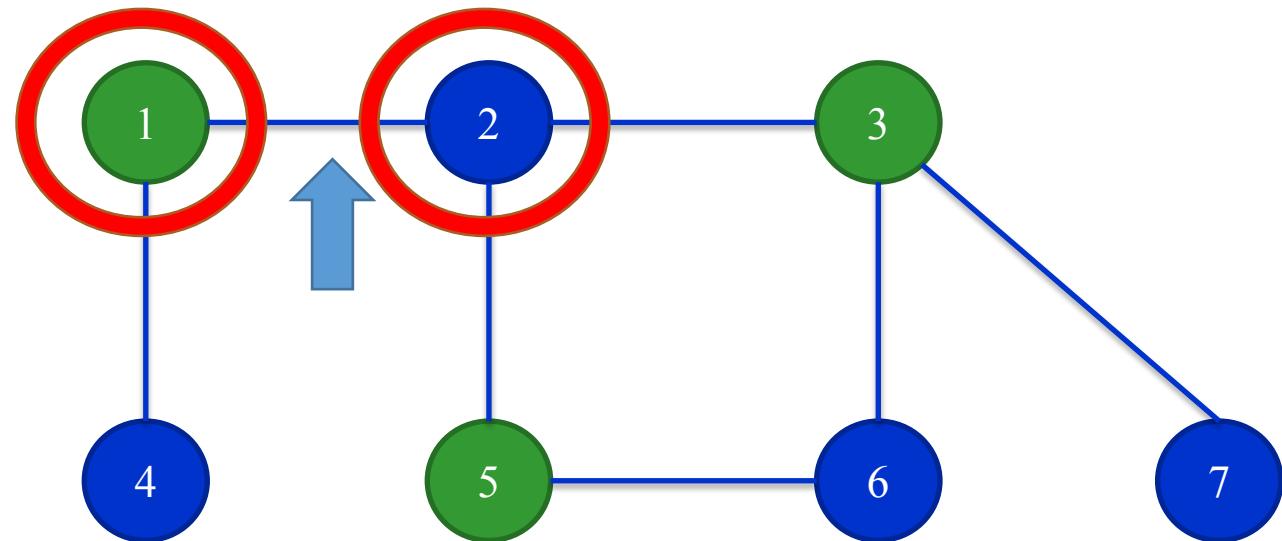
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*| > |A|$



# Approximation Algorithm

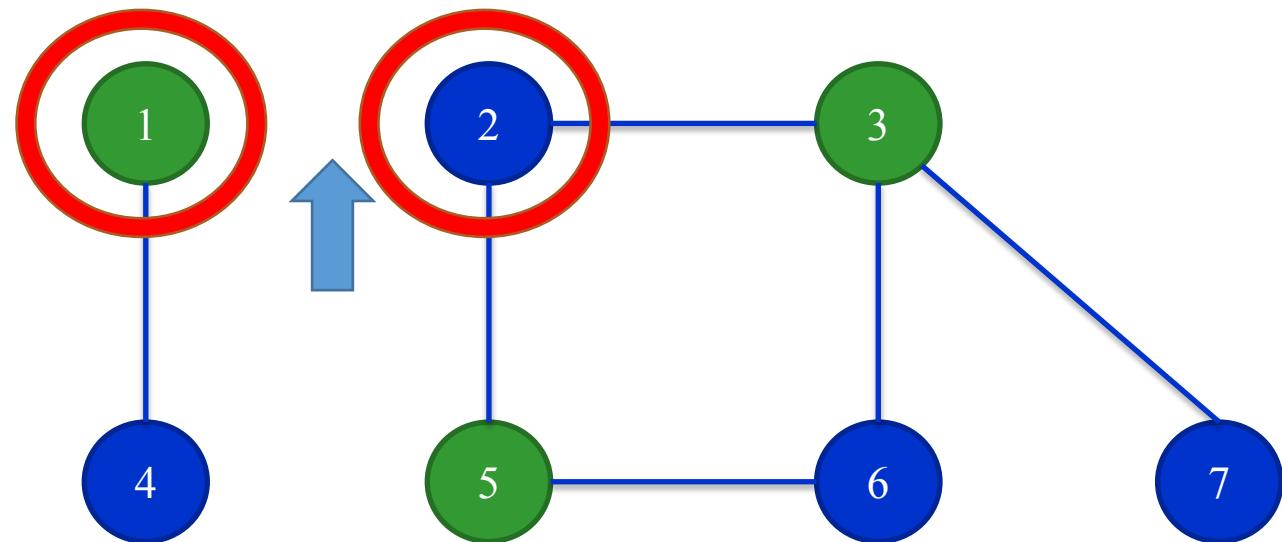
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*| > |A|$



# Approximation Algorithm

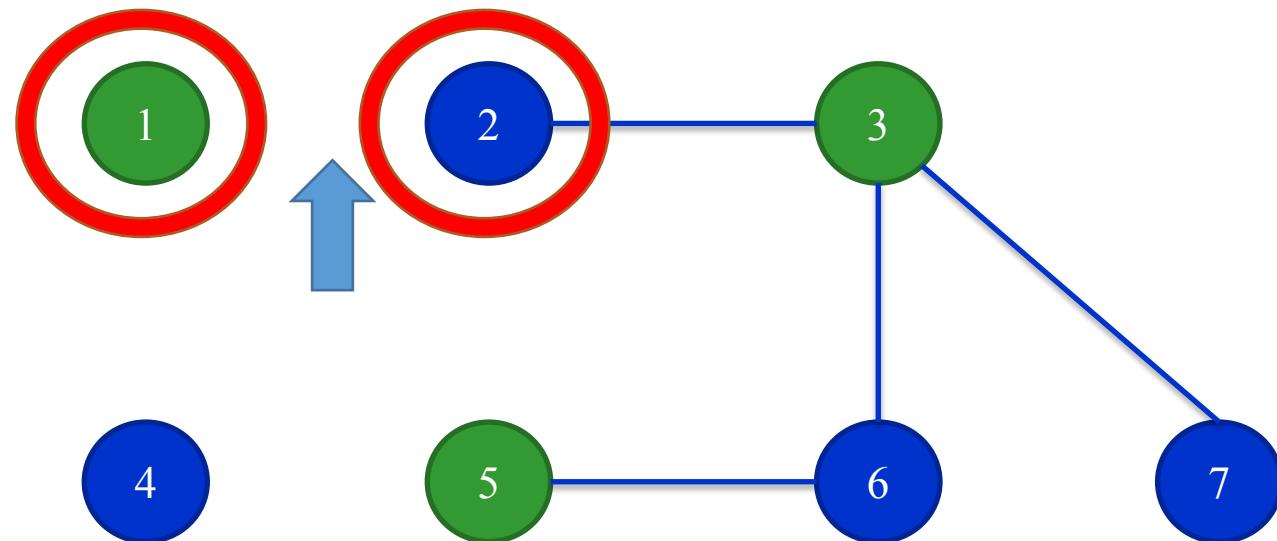
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*| > |A|$



# Approximation Algorithm

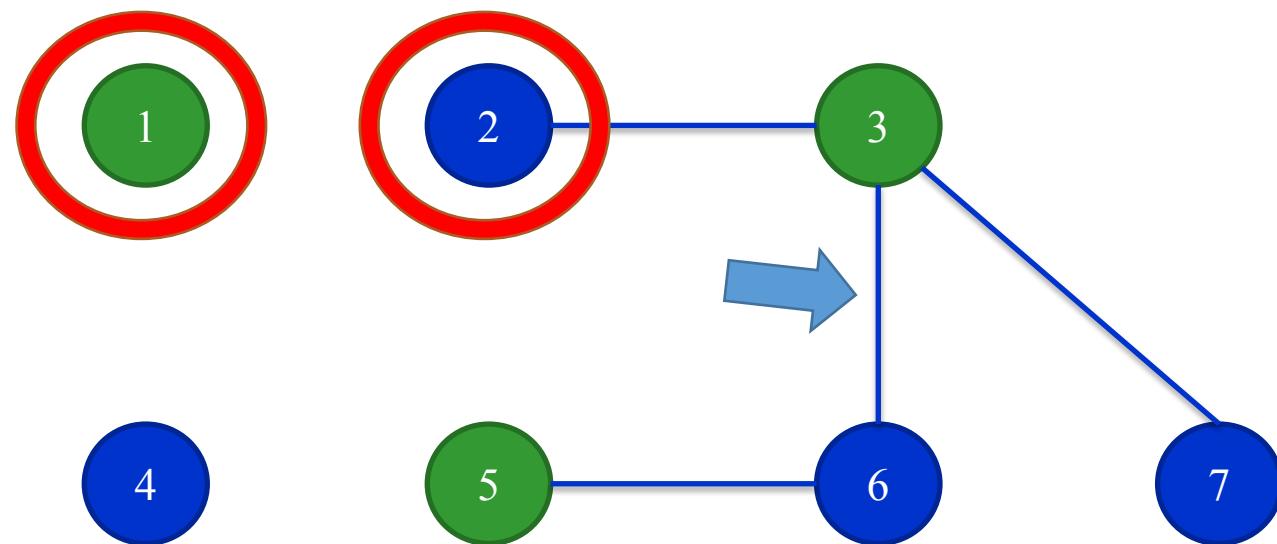
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*| > |A|$



# Approximation Algorithm

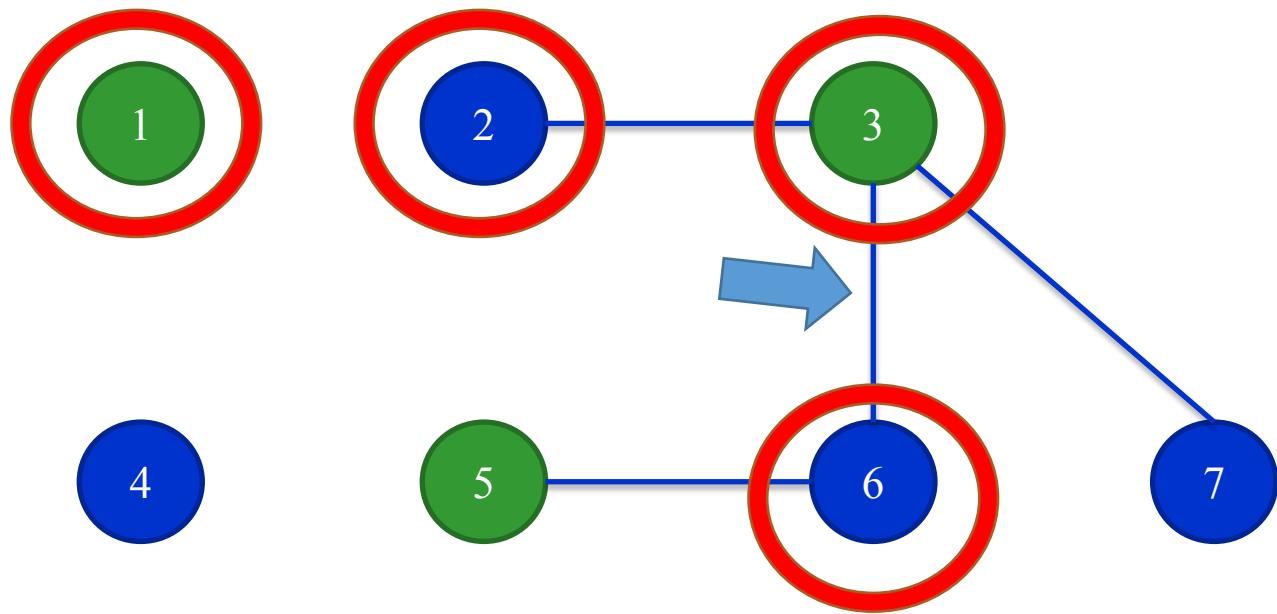
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*| > |A|$



# Approximation Algorithm

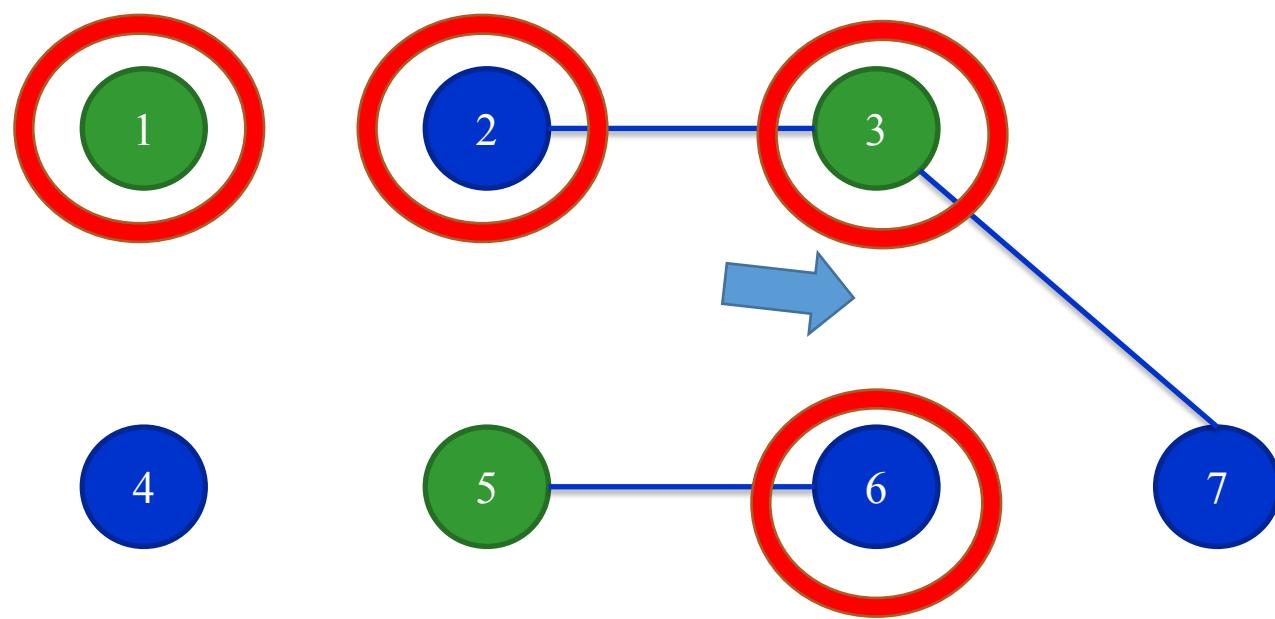
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*| > |A|$



# Approximation Algorithm

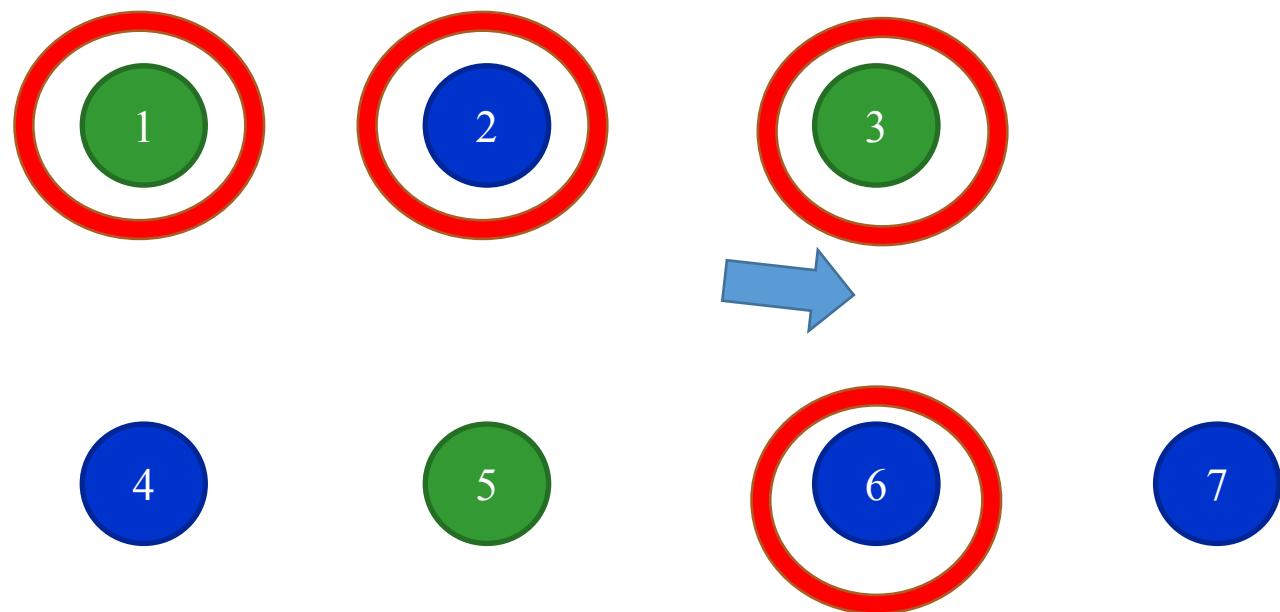
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

Example for  $|C^*| > |A|$



# Approximation Algorithm

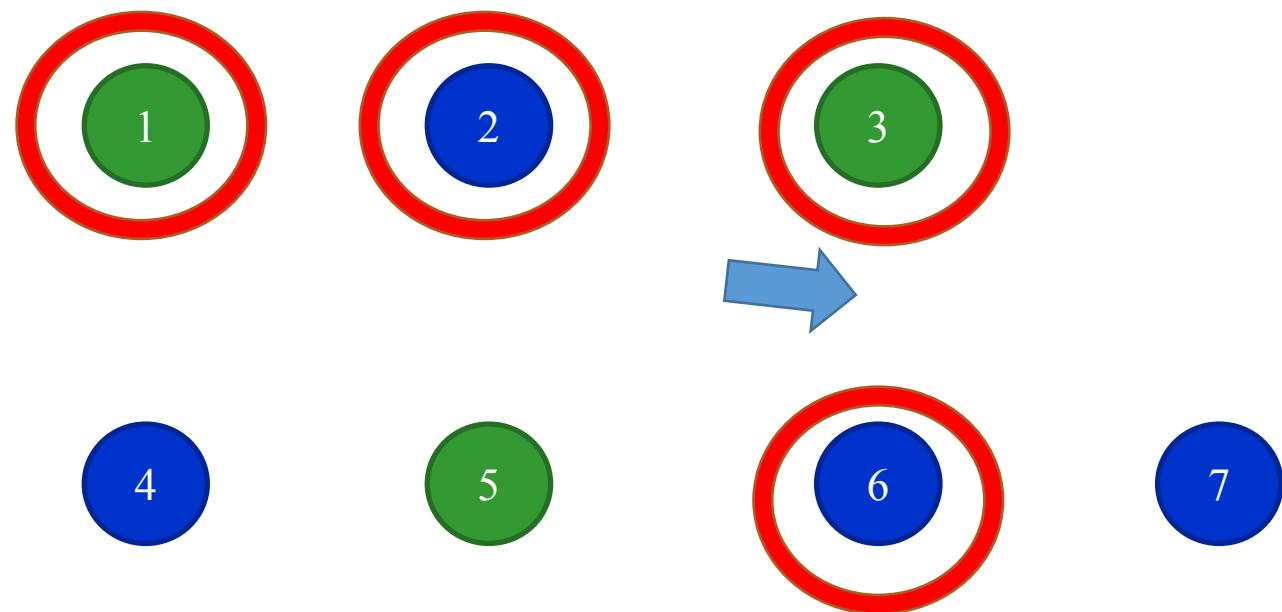
**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

$$1. \quad |C^*| \geq |A|$$

$$C^*=3$$

**Example for  $|C^*| > |A|$**



We selected 2 edges and managed to find 4 covers C, including the 3 optimal ones  $C^*$

# Approximation Algorithm

**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

1.  $|C^*| \geq |A|$  - A is the set of edges we selected during the algorithm
2.  $|C| = 2|A|$  - C is the number of vertex covers we found  
-  $C^*$  is the optimal number of vertex covers we found

# Approximation Algorithm

**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

1.  $|C^*| \geq |A|$  - A is the set of edges we selected during the algorithm
2.  $|C| = 2|A|$  - C is the number of vertex covers we found  
-  $C^*$  is the optimal number of vertex covers we found

For each edge we selected and placed in A we got two covers and placed in C

# Approximation Algorithm

**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

1.  $|C^*| \geq |A|$  - A is the set of edges we selected during the algorithm
2.  $|C| = 2|A|$  - C is the number of vertex covers we found  
-  $C^*$  is the optimal number of vertex covers we found
3.  $|C| \leq 2|C^*|$

# Approximation Algorithm

**Vertex-cover problem and a 2-approximation algorithm which runs in polynomial time.**

**Equations to consider:**

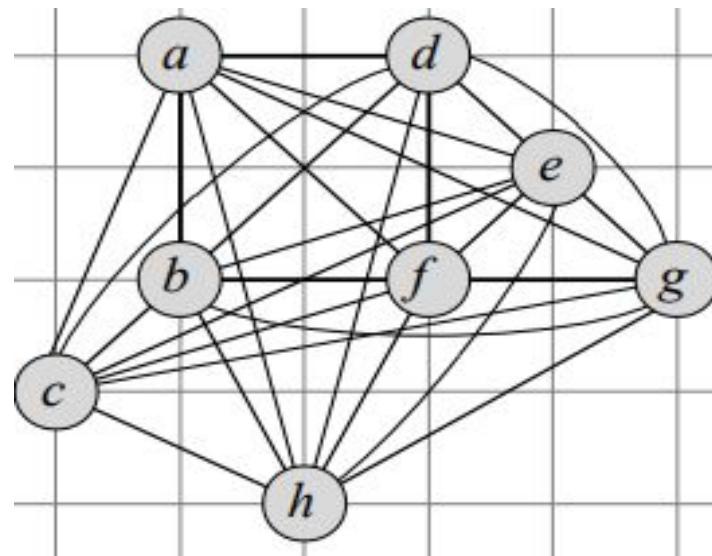
1.  $|C^*| \geq |A|$  - A is the set of edges we selected during the algorithm
2.  $|C| = 2|A|$  - C is the number of vertex covers we found  
-  $C^*$  is the optimal number of vertex covers we found
3.  $|C| \leq 2|C^*|$

# Travelling Salesman Problem

- It is possible to approximate it if :

$$c(u,w) \leq c(u,v) + c(v,w)$$

- This is possible in a 2D plane. Let us consider some cities in a dessert.

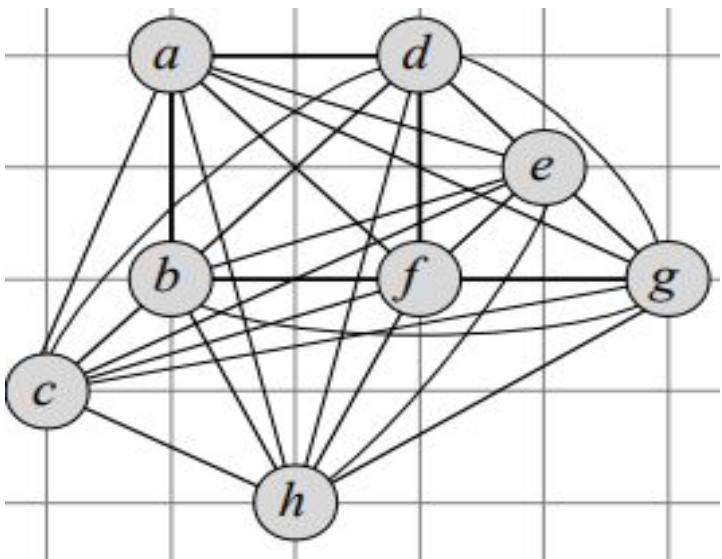


# Travelling Salesman Problem

- It is possible to approximate it if :

$$c(u,w) \leq c(u,v) + c(v,w)$$

- This is possible in a 2D plane. Let us consider some cities in a dessert.



Note that the curved edges are not referring to curved paths

Rather, it is referring to the st line distance

In this case, clearly triangle inequalities are maintained.

# The Approximation Algorithm

```
Approx-TSP(G, c)
```

```
    select r ∈ v[G];
```

```
    call MST-Prim(G,c,r) to construct MST with root r;
```

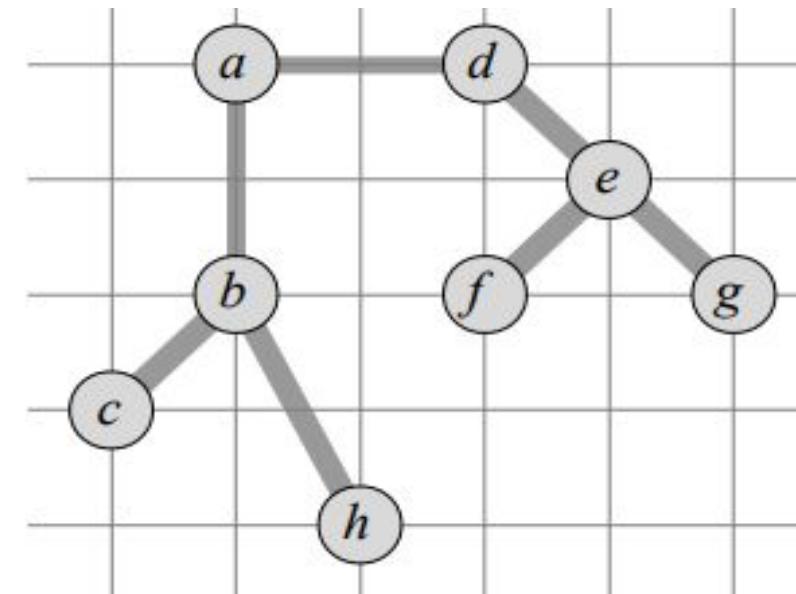
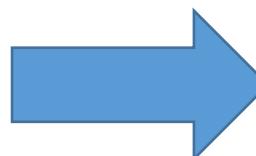
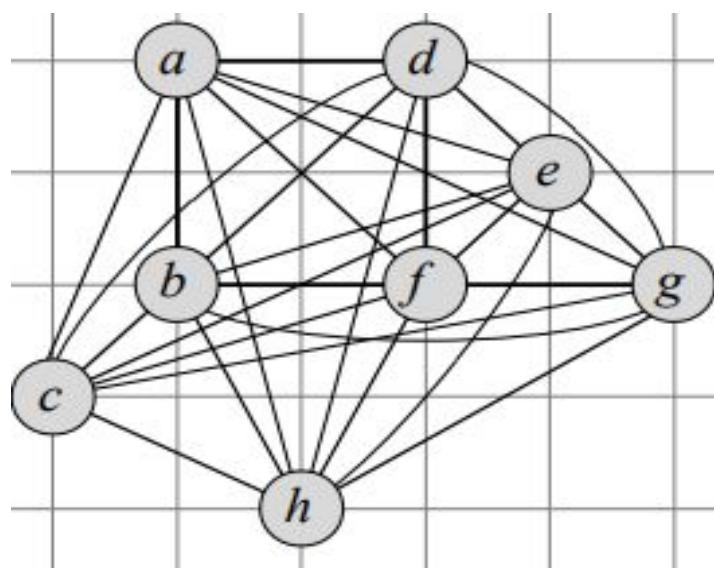
```
    let L = vertices on preorder walk of MST;
```

```
    let H = cycle that visits vertices in the order L;
```

```
return H
```

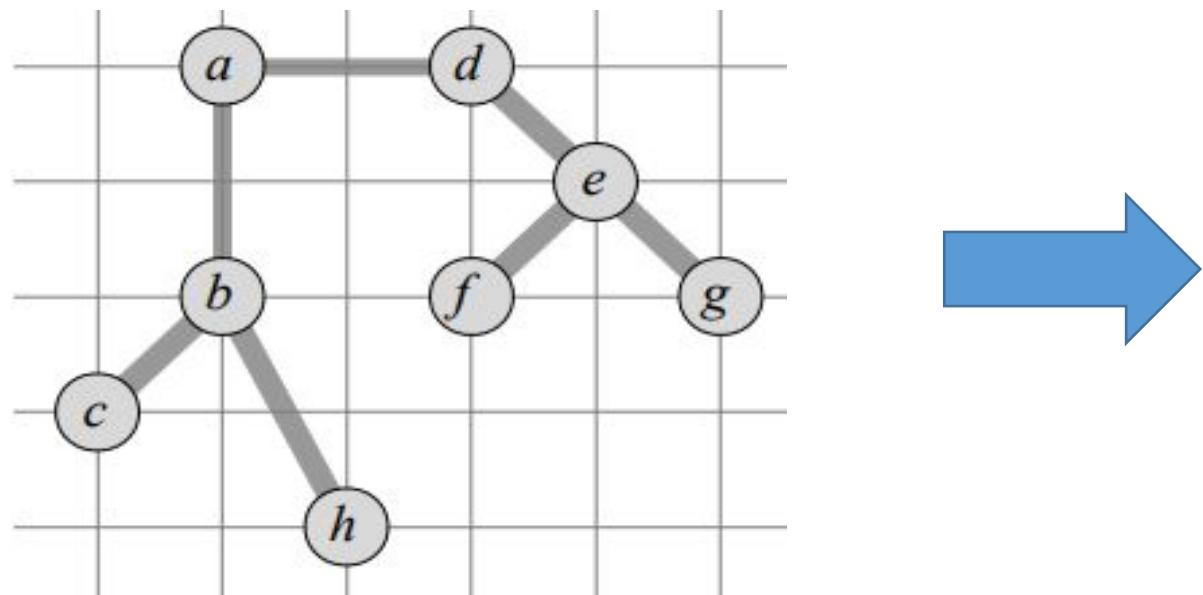
# Travelling Salesman Problem

- MST algorithm :Select a **root** and apply Prim's / Kruskal's algorithm



# Travelling Salesman Problem

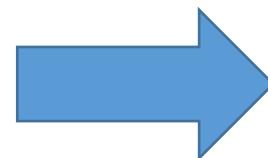
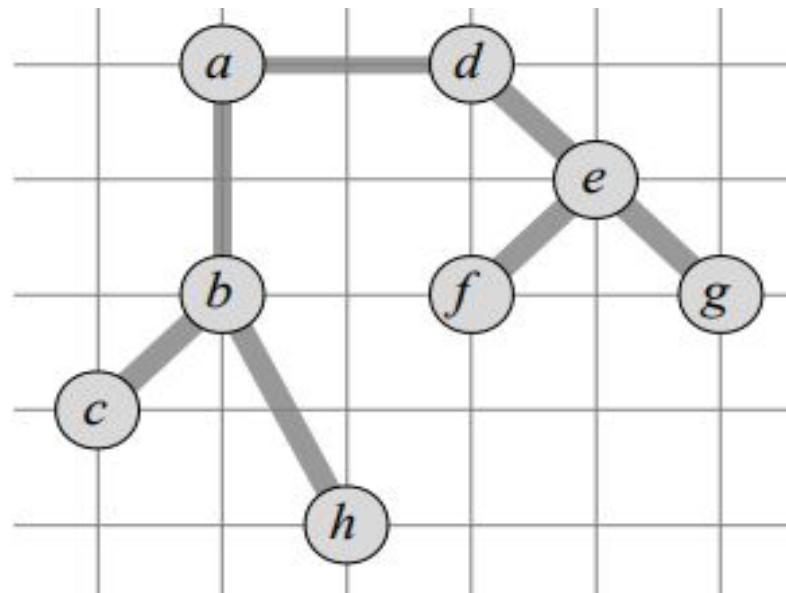
- MST algorithm



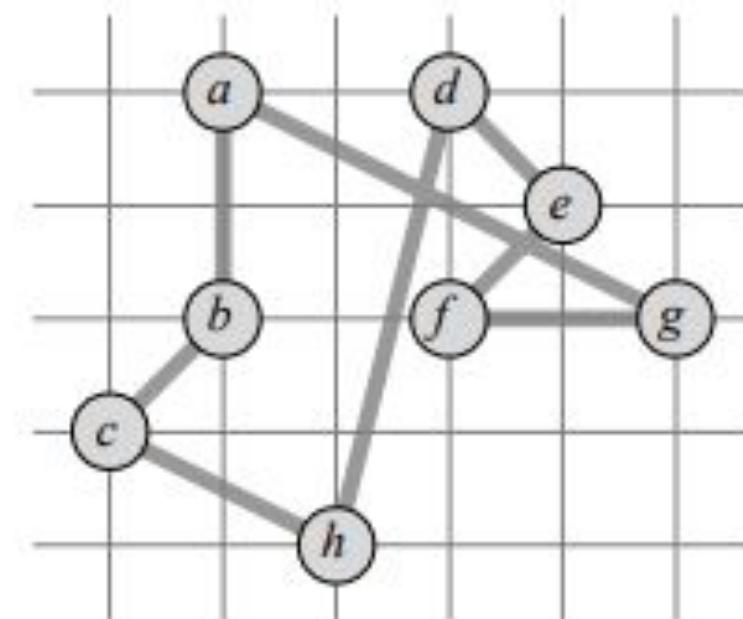
Pre-order traversal  
Sequence:  
{a,b,c,h,d,e,f,g,a}

# Travelling Salesman Problem

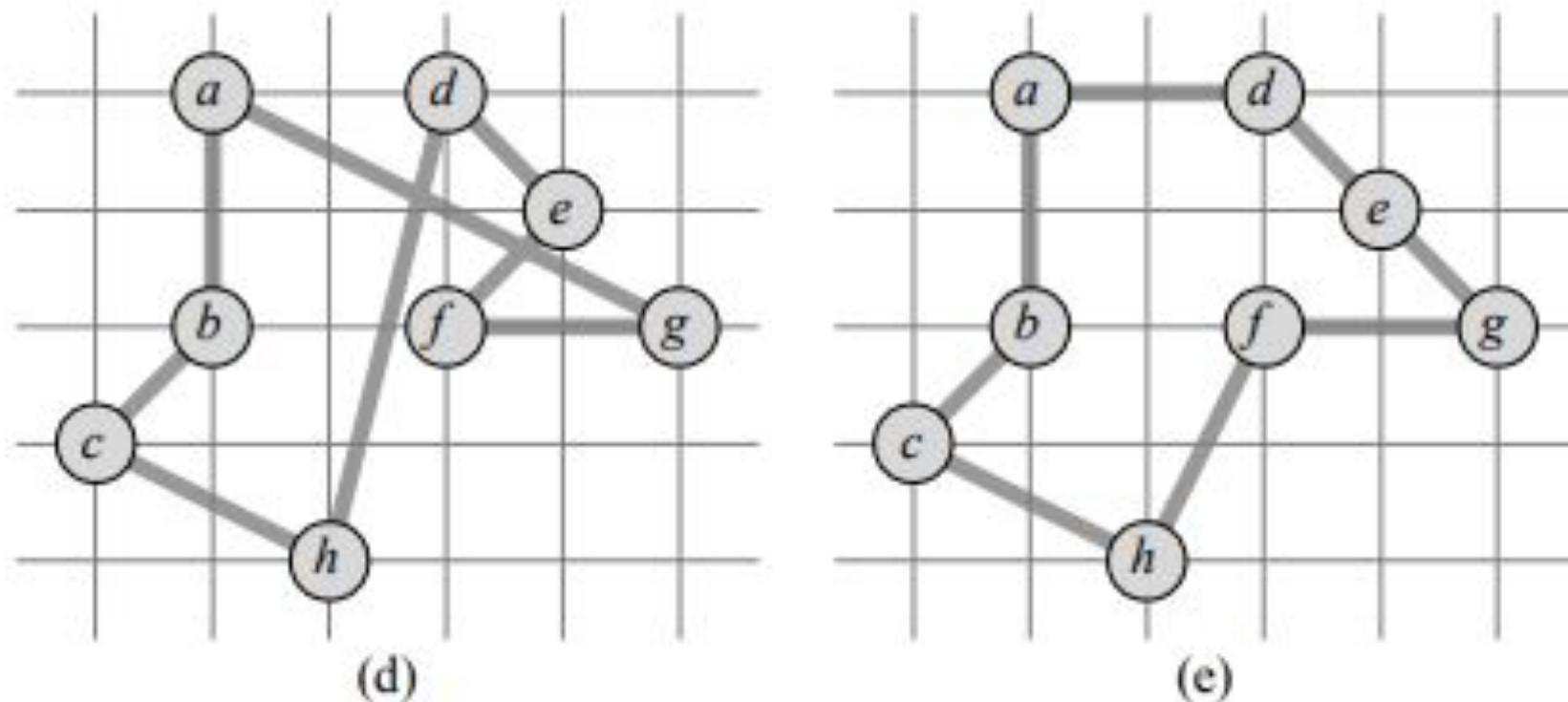
- MST algorithm



Pre-order traversal  
Sequence:  
 $\{a,b,c,h,d,e,f,g,a\}$

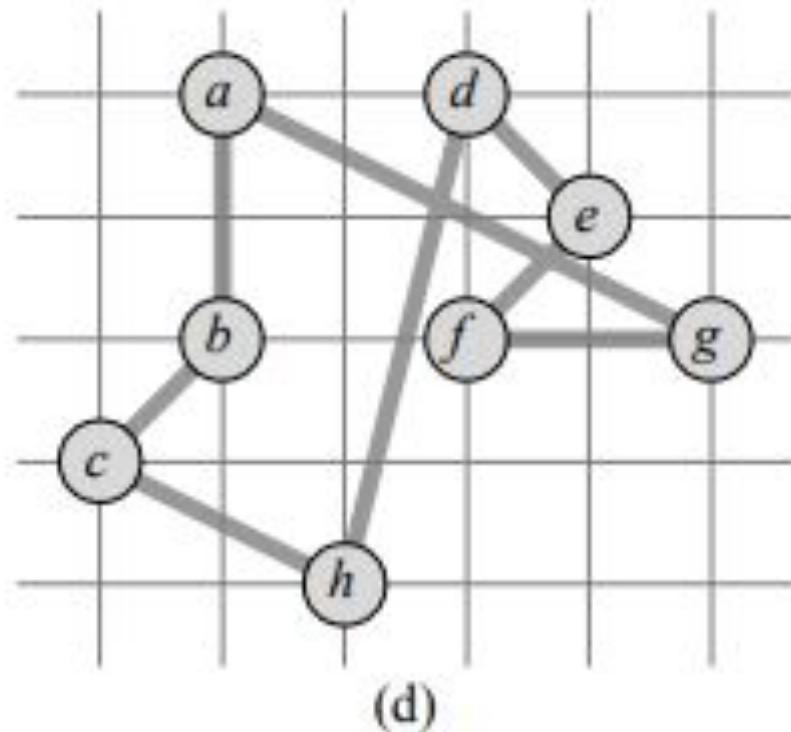


# Travelling Salesman Problem

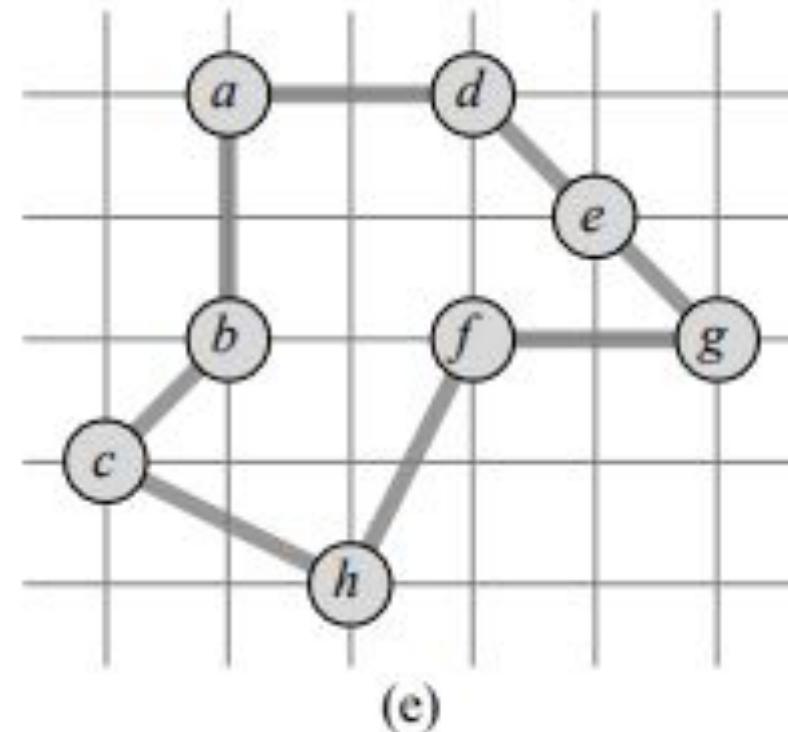


The preorder traversal sequence is  
the approximated solution

# Travelling Salesman Problem



The preorder traversal sequence is  
the approximated solution



This is the actual optimal solution

# Travelling Salesman Problem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality

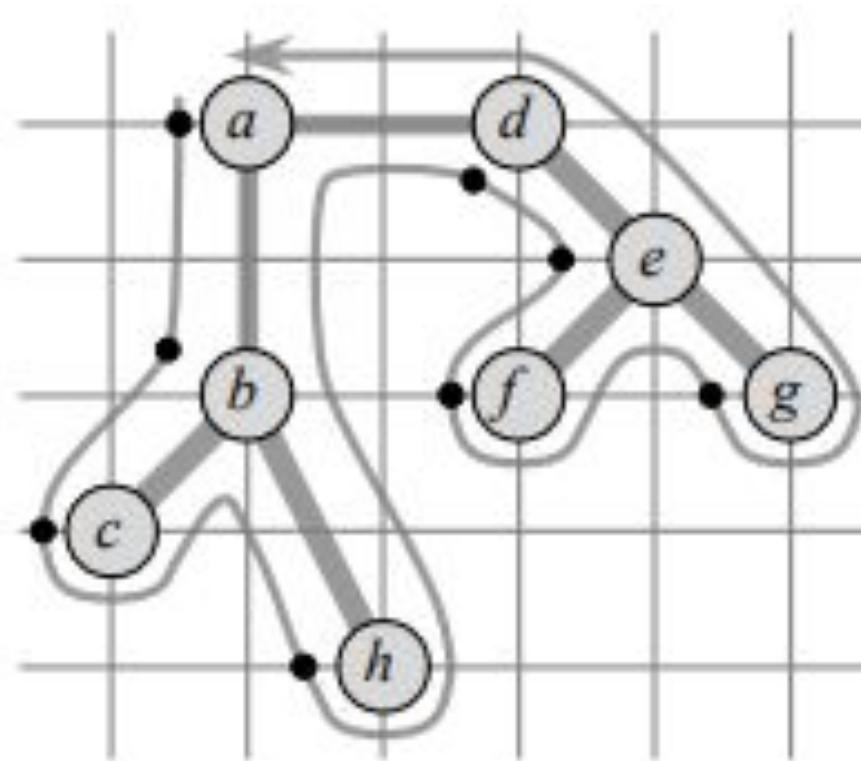
# Travelling Salesman Problem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality

In order to prove this, let us do a full walk  
On the graph

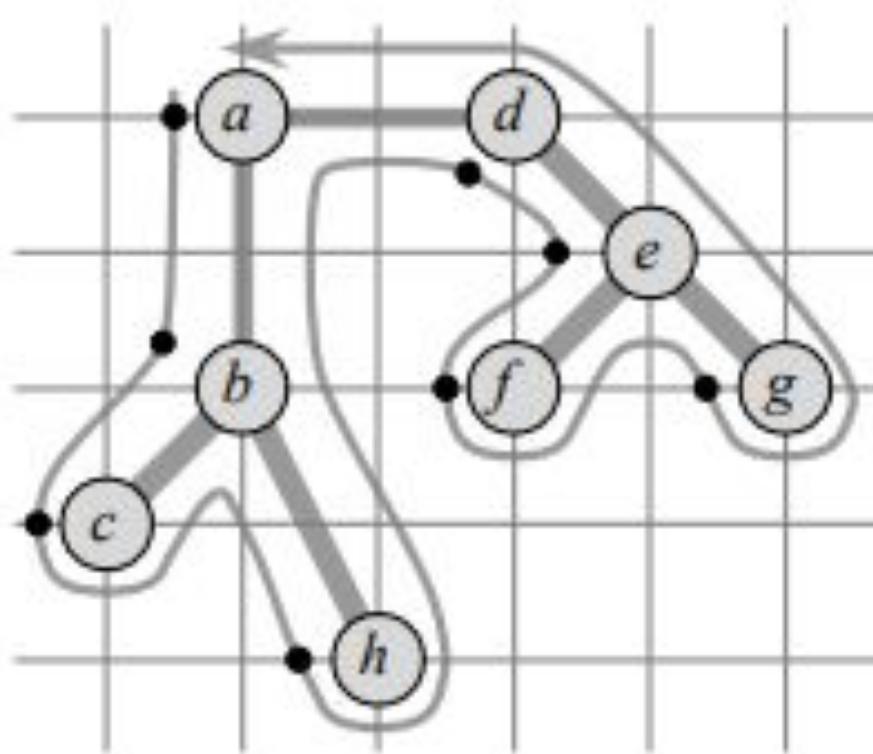
# Travelling Salesman Problem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality



# Travelling Salesman Problem

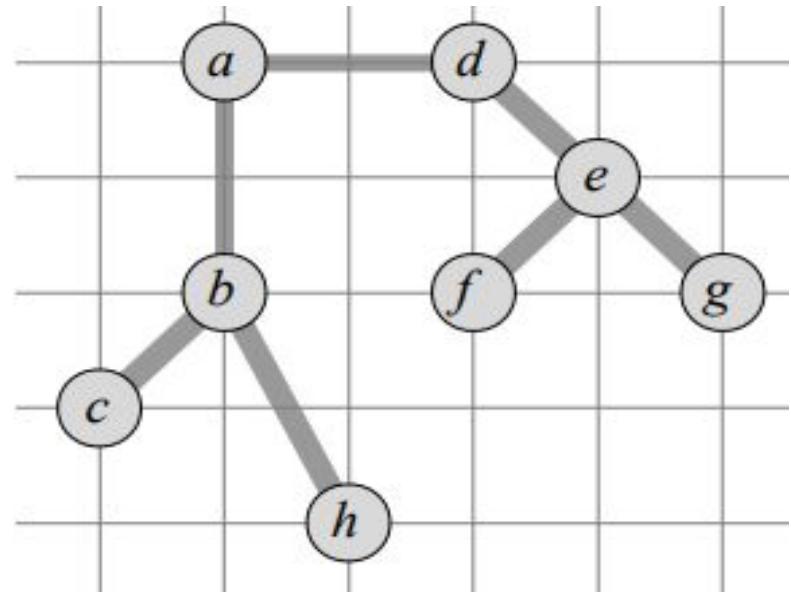
APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality



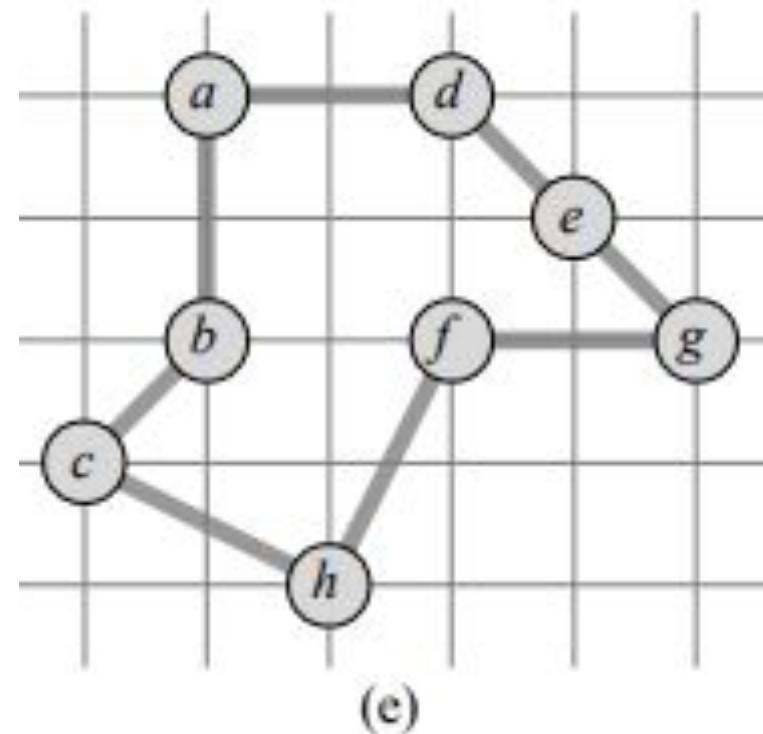
**Full walk:**  
{a; b; c; b; h; b; a; d; e; f; e; g;  
e; d; a}

# The Equations

$$C(T) \leq C(H^*)$$

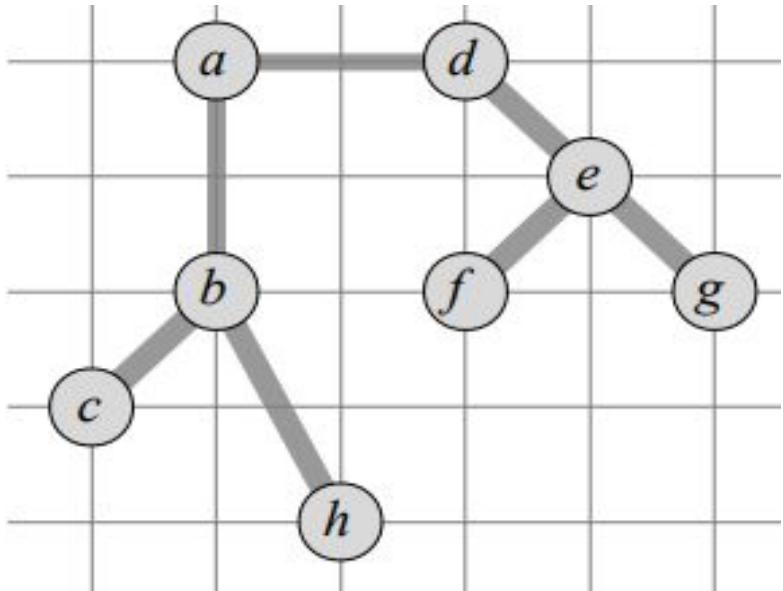


$\leq$

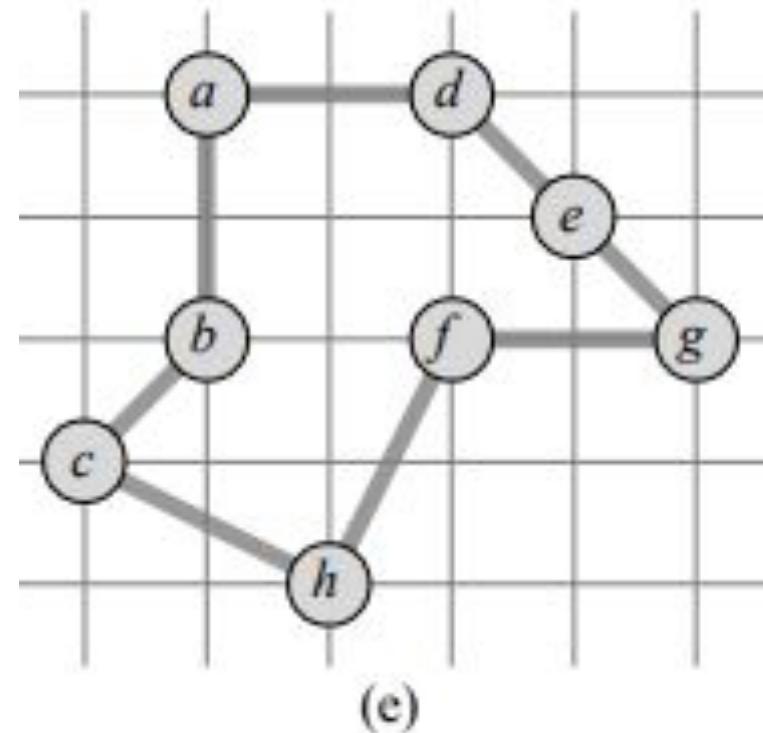


# The Equations

$$C(T) \leq C(H^*)$$



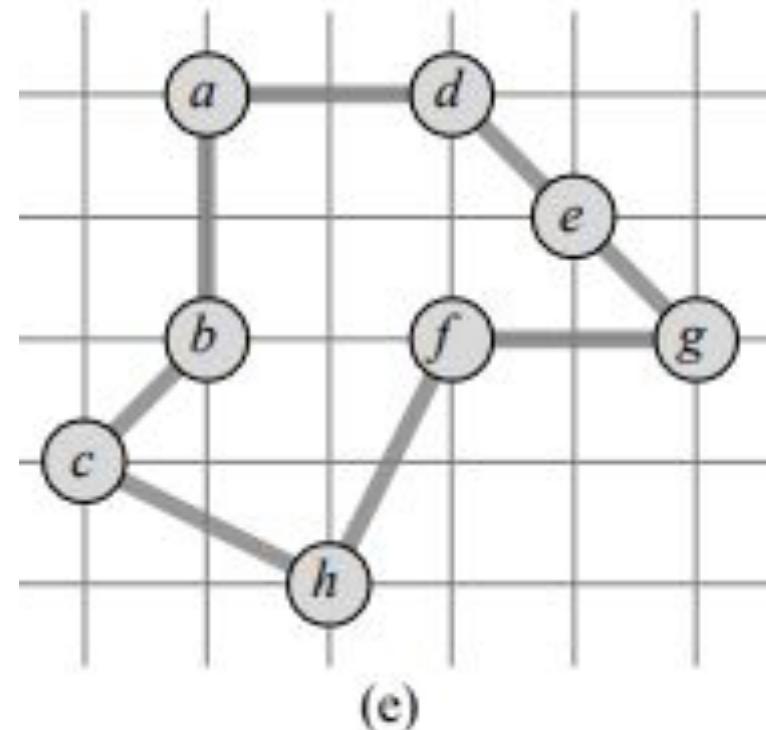
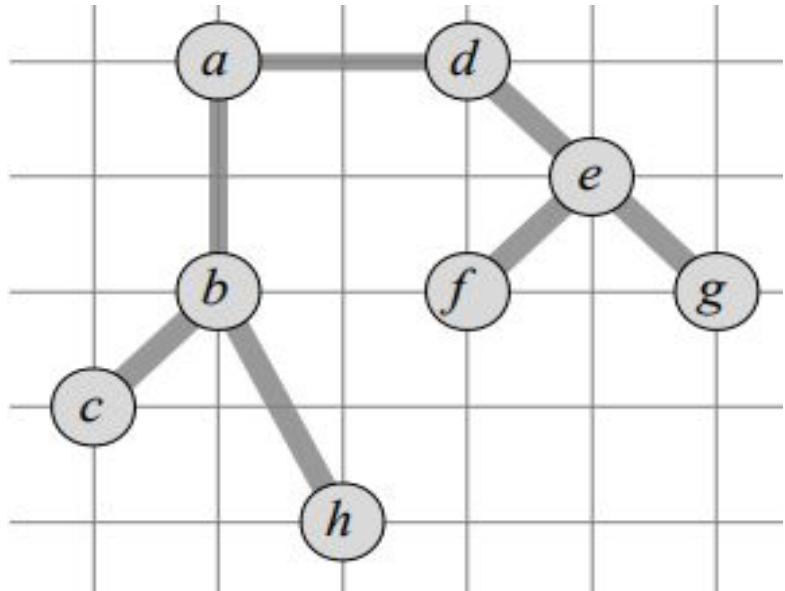
$\leq$



MST Has one less edge

# The Equations

$$C(T) \leq C(H^*)$$

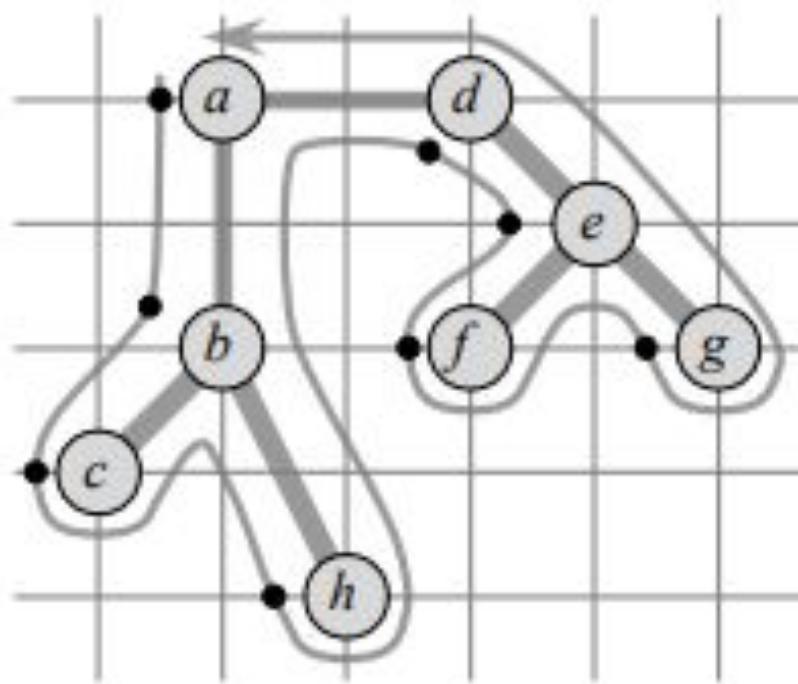


MST Has one less edge  
MST has the smallest edges

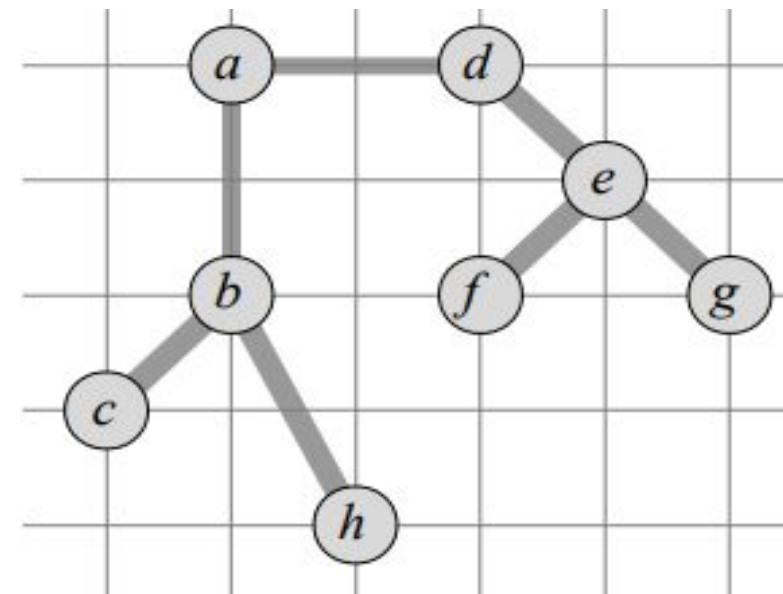
# The Equations

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$



=



# The Equations

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$

$$C(W) \leq 2C(H^*) \text{ (from the first two)}$$

# The Equations

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$

$$C(W) \leq 2C(H^*)$$

$$C(H) \leq C(W)$$

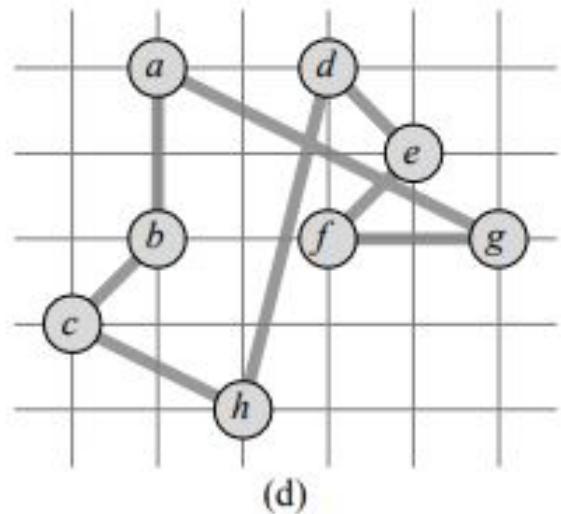
# The Equations

$$C(T) \leq C(H^*)$$

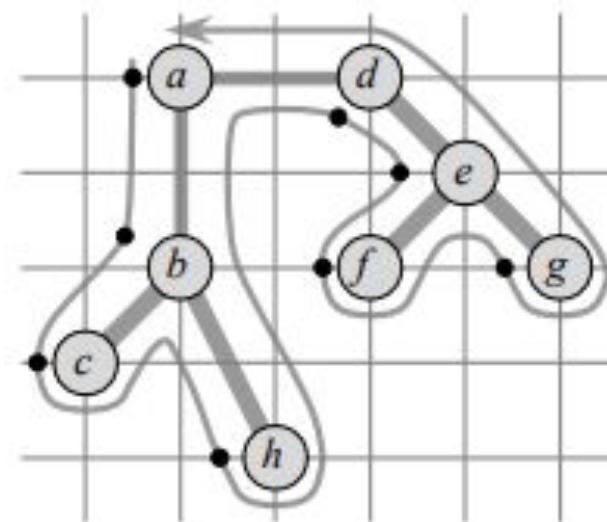
$$C(W) = 2C(T)$$

$$C(W) \leq 2C(H^*)$$

$$C(H) \leq C(W)$$



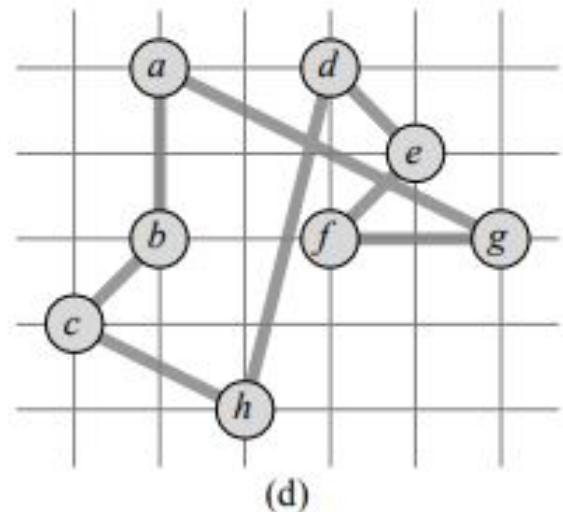
$\leq$



# The Equations

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$



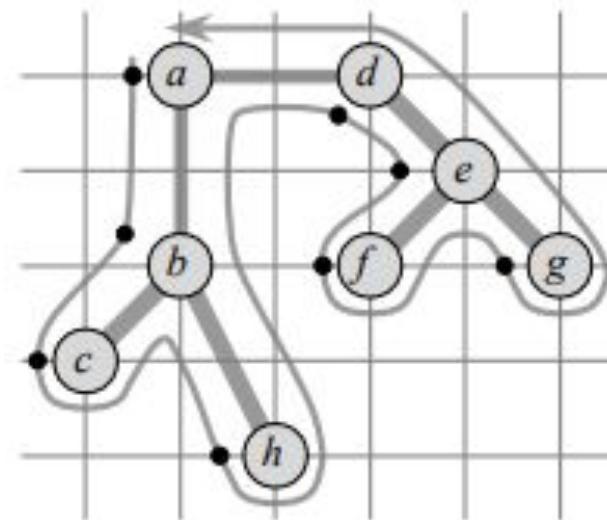
**The path we found**

Sequence: {a,b,c,h,d,e,f,g,a}

$$C(W) \leq 2C(H^*)$$

$$C(H) \leq C(W)$$

$\leq$



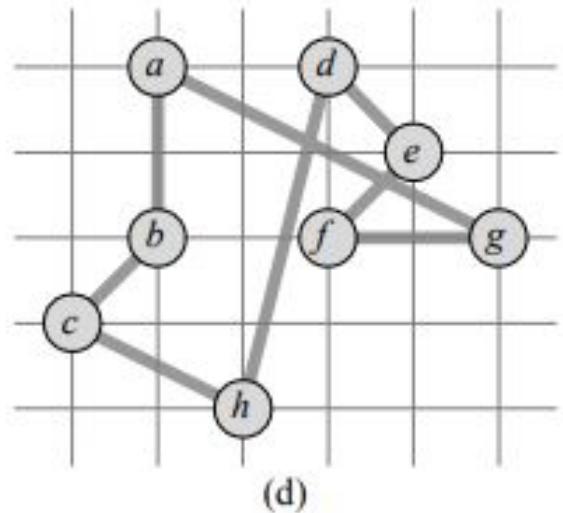
**Full walk:**

{a; b; c; b; h; b; a; d; e; f; e; g; e; d; a}

# The Equations

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$



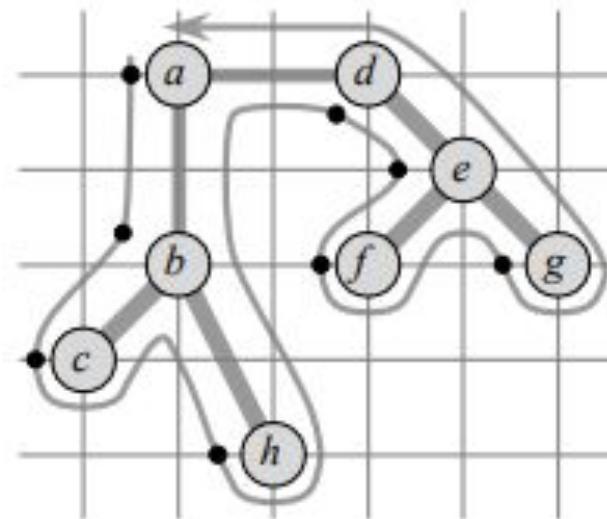
**The path we found**

Sequence: {a,b,c,h,d,e,f,g,a}

$$C(W) \leq 2C(H^*)$$

$$C(H) \leq C(W)$$

$\leq$



**Full walk:**

{**a**; **b**; **c**; **b**; **h**; **b**; **a**; **d**; **e**; **f**; **e**; **g**; **e**; **d**; **a**}

In the path  
we found  
We went from  
C to h with less  
cost

(Due to  
Triangle inequality)

# The Equations

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$

$$C(W) \leq 2C(H^*)$$

$$C(H) \leq C(W)$$

**$C(H) \leq 2C(H^*)$  – from the last two**

# The Equations

$$C(T) \leq C(H^*)$$

$$C(W) = 2C(T)$$

$$C(W) \leq 2C(H^*)$$

$$C(H) \leq C(W)$$

**$C(H) \leq 2C(H^*)$  – from the last two  
(proved)**

THE END