

# Module IV

## Complexity Theory

### NP Hard and NP Complete Problems

#### Decision problems vs. optimization problems

The problems we are trying to solve are basically of two kinds. In **decision problems** we are trying to decide whether a statement is true or false. In optimization problems we are trying to find the solution with the best possible score according to some scoring scheme. **Optimization problems** can be either maximization problems, where we are trying to maximize a certain score, or minimization problems, where we are trying to minimize a cost function.

#### Example 1: Hamiltonian cycles

Given a directed graph, we want to decide whether or not there is a Hamiltonian cycle in this graph. This is a decision problem.

## Example 2: TSP - The Traveling Salesman Problem

Given a complete graph and an assignment of weights to the edges, find a Hamiltonian cycle of minimum weight. This is the optimization version of the problem. In the decision version, we are given a weighted complete graph and a real number  $c$ , and we want to know whether or not there exists a Hamiltonian cycle whose combined weight of edges does not exceed  $c$ .

Each optimization problem has a corresponding decision problem.

### **Deterministic and non-deterministic algorithms**

In the context of programming, an Algorithm is a set of well-defined instructions in sequence to perform a particular task and achieve the desired output. Here we say set of defined instructions which means that somewhere user knows the outcome of those instructions if they get executed in the expected manner.

On the basis of the knowledge about outcome of the instructions, there are two types of algorithms namely – Deterministic and Non-deterministic Algorithms.

A deterministic algorithm is an algorithm that is purely determined by its inputs, where no randomness is involved in the model. **Deterministic algorithms will always come up with the same result given the same inputs. Algorithms such that the result of every operation is uniquely defined are called *deterministic algorithms*.**

Up to now, we have focused on problems that were deterministic.

A non-deterministic algorithm can provide different outputs for the same input on different executions. Unlike a deterministic algorithm which produces only a single output for the same input even on different runs, a non-deterministic algorithm travels in various routes to arrive at the different outcomes.

Non-deterministic algorithms are useful for finding approximate solutions, when an exact solution is difficult or expensive to derive using a deterministic algorithm.

Discussing **nondeterministic algorithms** requires three new functions:

1. **Choice(S)**: arbitrarily chooses one of the elements of set S

2.**Failure()** signals an unsuccessful completion

3.**Success()** signals a successful completion

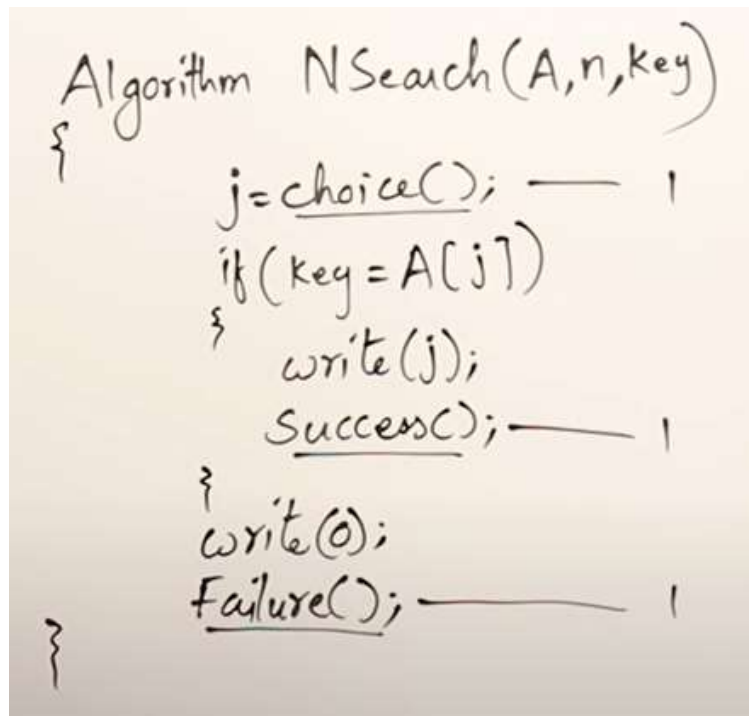
*A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a successful signal.*

### ***Nondeterministic search***

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

The computing times for **Choice**, **Success**, and **Failure** are taken to be  $O(1)$ .

Eg.



```
Algorithm NSearch(A, n, key)
{
    j = choice(); — 1
    if (key = A[j])
    {
        write(j);
        Success(); — 1
    }
    write(0);
    Failure(); — 1
}
```

$J = \text{Choice}()$  gives the index of the key element. If element in  $j$ th index is key element then search is successful. Otherwise failure.

How this choice came to know that key element is present in that  $j$ th position. That's why it is non deterministic. If we are able to find the key position in constant time then we can fill up this choice() function.

For example,

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

If we know the key is at position 3 (7.0) in a constant time it will become deterministic. That is algorithm directly gets the answer. (method is not known now, may be developed in future)

A machine capable of executing a nondeterministic algorithm in this way is called a *nondeterministic machine*.

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
1	Definition	The algorithms in which the result of every algorithm is uniquely defined are known as the	On other hand, the algorithms in which the result of every algorithm is not uniquely defined and

<b>Sr. No.</b>	<b>Key</b>	<b>Deterministic Algorithm</b>	<b>Non-deterministic Algorithm</b>
		<p>Deterministic Algorithm. In other words, we can say that the deterministic algorithm is the algorithm that performs fixed number of steps and always get finished with an accept or reject state with the same result.</p>	<p>result could be random are known as the Non-Deterministic Algorithm.</p>
2	Execution	<p>In Deterministic Algorithms execution, the target machine executes the same instruction and results</p>	<p>On other hand in case of Non-Deterministic Algorithms, the machine executing each operation is</p>

<b>Sr. No.</b>	<b>Key</b>	<b>Deterministic Algorithm</b>	<b>Non-deterministic Algorithm</b>
		same outcome which is not dependent on the way or process in which instruction get executed.	allowed to choose any one of these outcomes subjects to a determination condition to be defined later.
3	Type	On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for a particular input instructions the machine will give always the same output.	On other hand Non deterministic algorithm are classified as non-reliable algorithms for a particular input the machine will give different output on different executions.



Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
4	Execution Time	As outcome is known and is consistent on different executions so Deterministic algorithm takes <b>polynomial time</b> for their execution.	On other hand as outcome is not known and is non-consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time.
5	Execution path	In deterministic algorithm the path of execution for algorithm is same in every execution.	On other hand in case of Non-Deterministic algorithm the path of execution is not same for algorithm in every execution and could take

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
			any random path for its execution.

## Polynomial time and exponential time algorithms

Basically, we have problems that can be solved in a **short time (polynomial)** and ones that **require vast amounts of time** because of the numerous possibilities inherent in the problems (**non polynomial**). The first group consists of problems whose solution times are bounded by polynomials of small degree.

Eg.  $O(\log n)$ ,  $O(n)$

The second group is made up of problems whose best-known algorithms are non polynomial.

Eg.  $O(n^2 2^n)$ ,  $2^{n/2}$

Examples are given here.

<u>Polynomial Time</u>	<u>Exponential Time</u>
Linear Search — $n$	0/1 Knapsack — $2^n$
Binary Search — $\log n$	Traveling SP — $2^n$
Insertion Sort — $n^2$	Sum of Subsets — $2^n$
Merge Sort — $n \log n$	Graph Coloring — $2^n$
Matrix Multiplication — $n^3$	Hamiltonian Cycle — $2^n$

Exponential time taking algorithms are much bigger than Polynomial time taking algorithms

So we want polynomial time algorithms for solving these exponential time algorithms.

## Classes of Algorithms

### Class P

**P** is a set of all **decision problems** solvable by **deterministic algorithms** in **polynomial time**.

The class P EXAMPLE: The Minimum Spanning Tree Problem is in the class P.

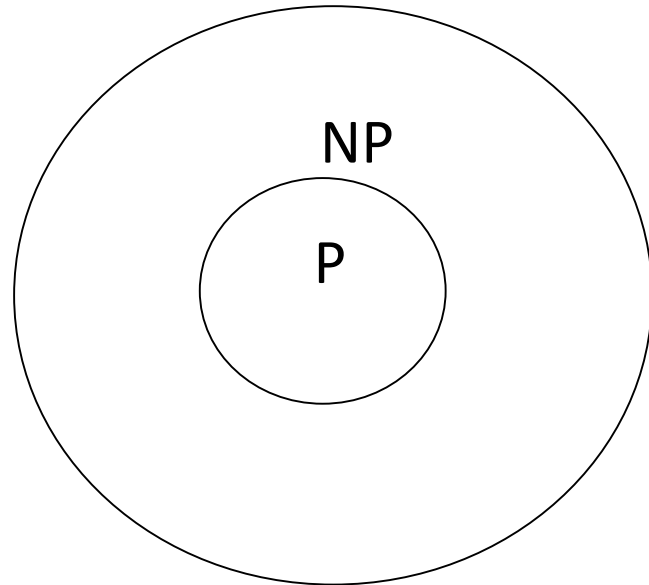
### The class NP

A problem that is NP-Complete has the property that it can be solved in polynomial time iff all other NP-Complete problems can also be solved in polynomial time. **If an NP-Hard problem can be solved in polynomial time, then all NP-Complete problems can be solved in polynomial time. All NP-Complete problems are NP-Hard, but not all NP-Hard problems are NP-Complete.**

**NP** stands for **Nondeterministic Polynomial**

NP is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Deterministic algorithms are the special case of non deterministic one. The



given figure shows the relationship between P and NP.

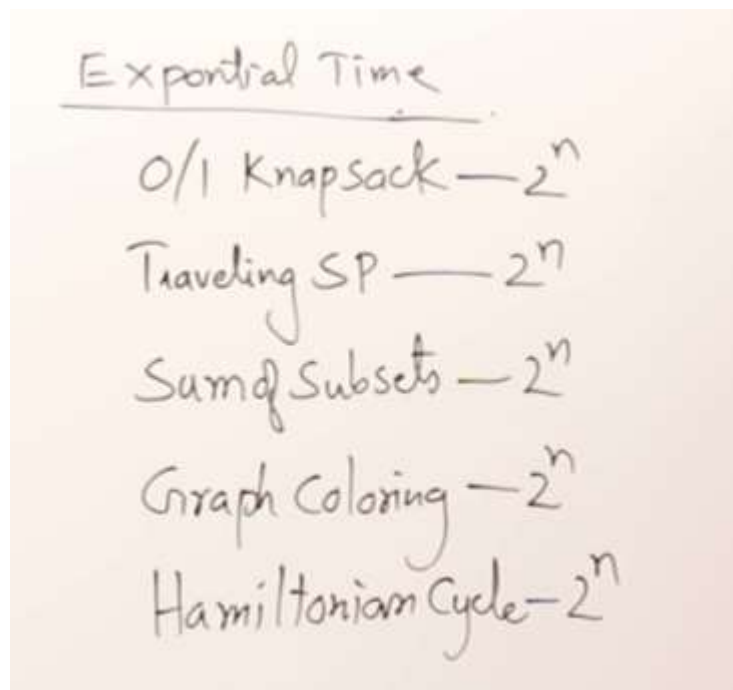
## The P=NP Problem

It is not hard to show that every problem in P is also in NP, but it is unclear whether every problem in NP is also in P. The P=NP Problem. The best we can say is that thousands of computer scientists have been unsuccessful for

decades to design polynomial-time algorithms for some problems in the class NP. This constitutes overwhelming empirical evidence that the classes P and NP are indeed distinct, but no formal mathematical proof of this fact is known.

### **The Satisfiability problem (CNF-Satisfiability)**

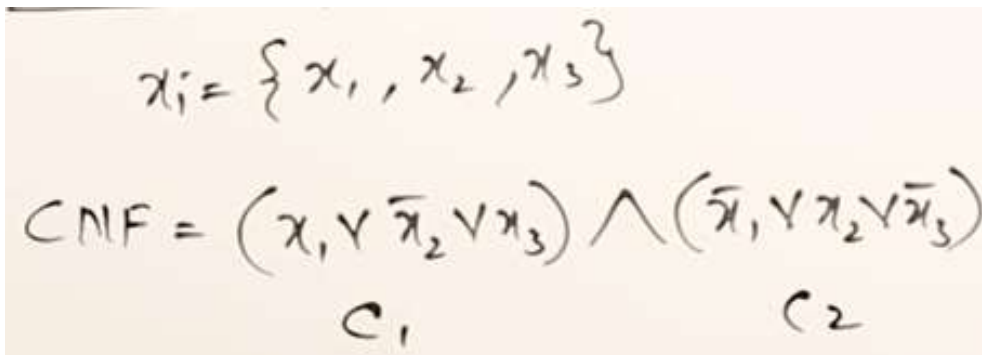
If one problem is solved in polynomial time in the given example it is easy to solve other problems also in polynomial time.



To do this, we take satisfiability problem as base problem.

- An expression  $E$  is *satisfiable* if there exists a truth assignment to the variables in  $E$  that makes  $E$  true.
- The *satisfiability problem* (SAT) is to determine whether a given boolean expression is satisfiable.
- SAT can be used to prove that other problems are NP complete by showing that the other problem is in NP and that SAT can be reduced to the other problem in polynomial time.

For example



The image shows handwritten mathematical expressions on a light orange background. The first line defines a set of variables:  $x_i = \{x_1, x_2, x_3\}$ . The second line shows a Conjunctive Normal Form (CNF) formula:  $CNF = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$ . Below the first clause,  $(x_1 \vee \bar{x}_2 \vee x_3)$ , is the label  $C_1$ . Below the second clause,  $(\bar{x}_1 \vee x_2 \vee \bar{x}_3)$ , is the label  $C_2$ .

**The satisfiability problem is to find out in what values of  $x_i$  this formula is true.**

This is CNF propositional calculus formula.

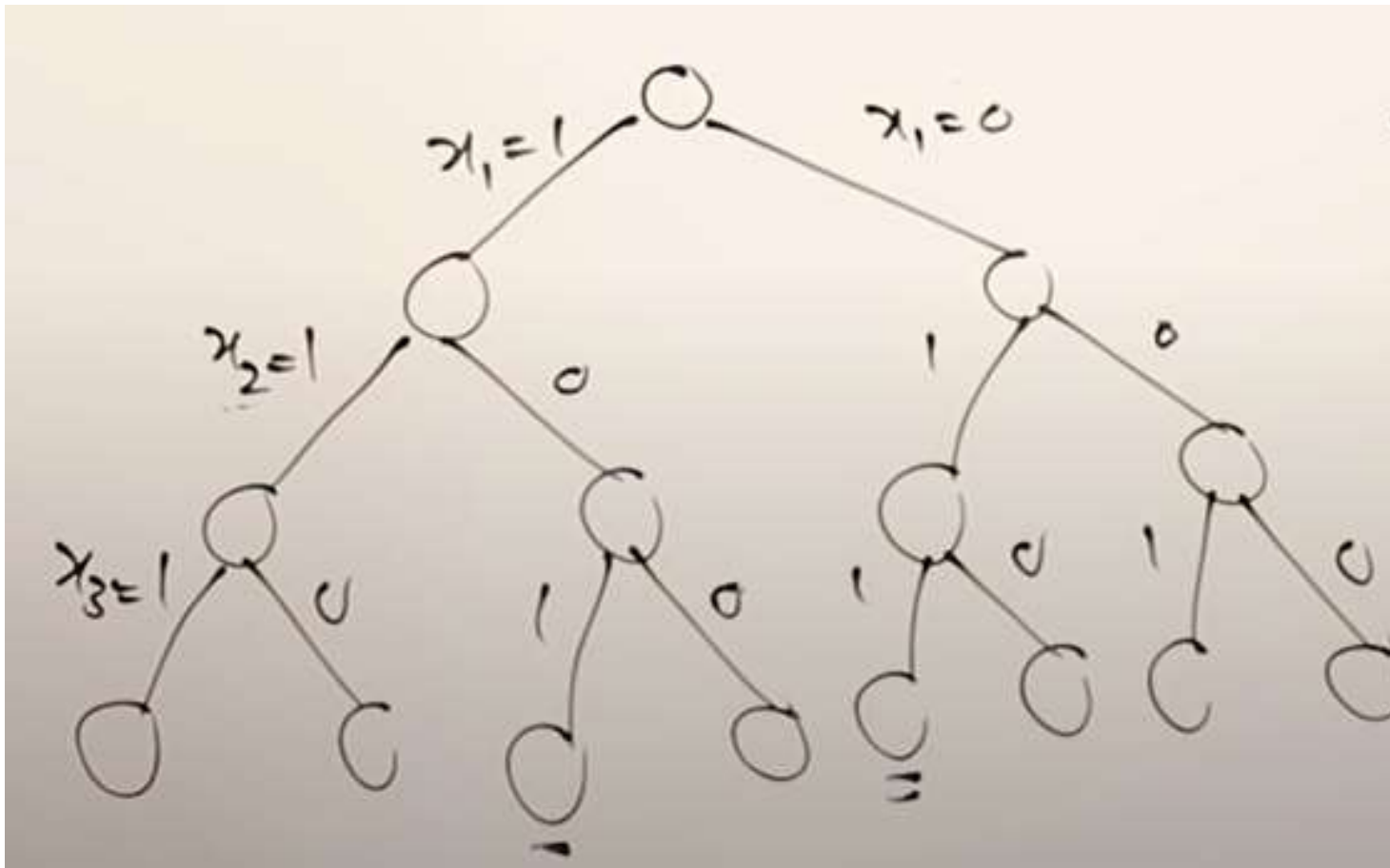
The possible values of  $x_i$  are:

x1	x2	x3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Example : check the value 8  $\rightarrow 2^3 \rightarrow 2^n$  – **exponential time taking problem**

State space tree of the above values are:





If **satisfiability solved in polynomial time** all the exponential time algorithms can be solved in polynomial time. Satisfiability problem can be solved in polynomial time.

For example 0/1 Knapsack problem,  $p=\{10,2,3\}$  and  $w=\{5,4,3\}$ ,  $m=8$  and  $n=3$ . This can be represented by Boolean variables such as

$$x_i = \{0/1, 0/1, 0/1\}$$

x1 x2 x3

0 0 0

0 0 1

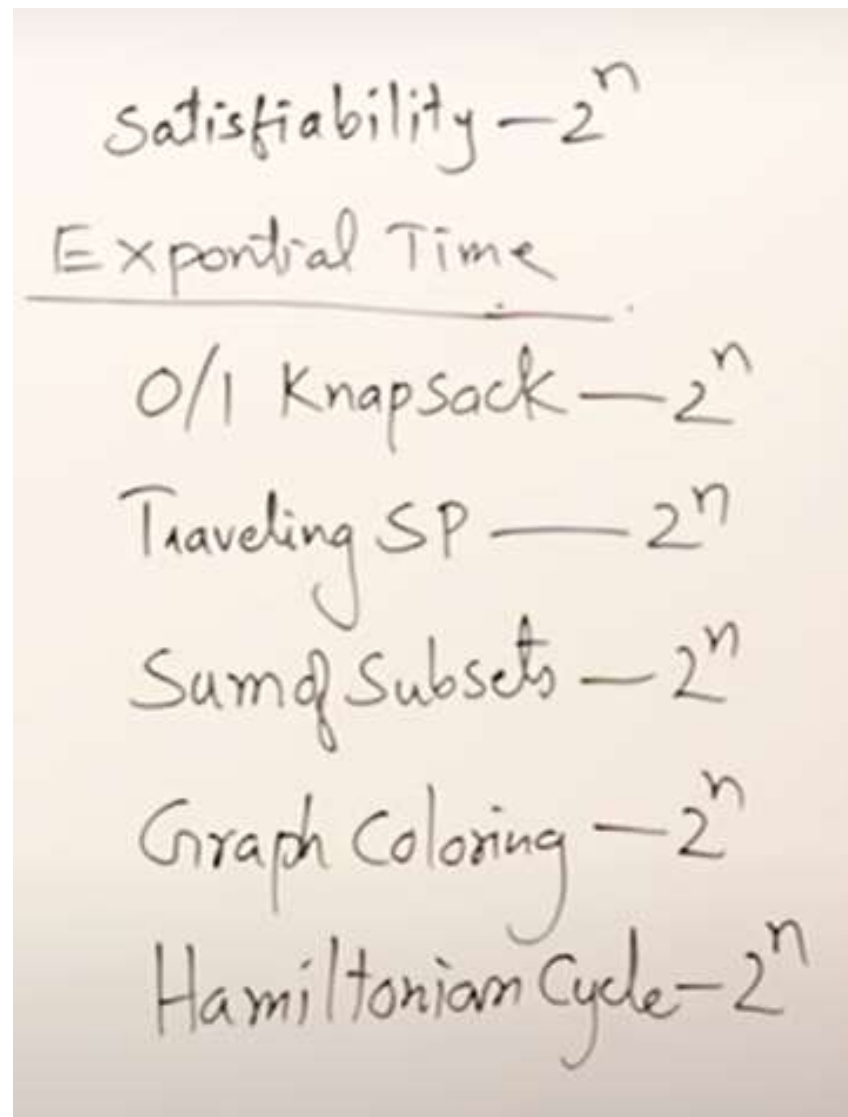
0 1 0

.....

.....

n=3, so  $2^3 \rightarrow 2^n$

0/1 Knapsack problem can also be solved using the above state space tree.



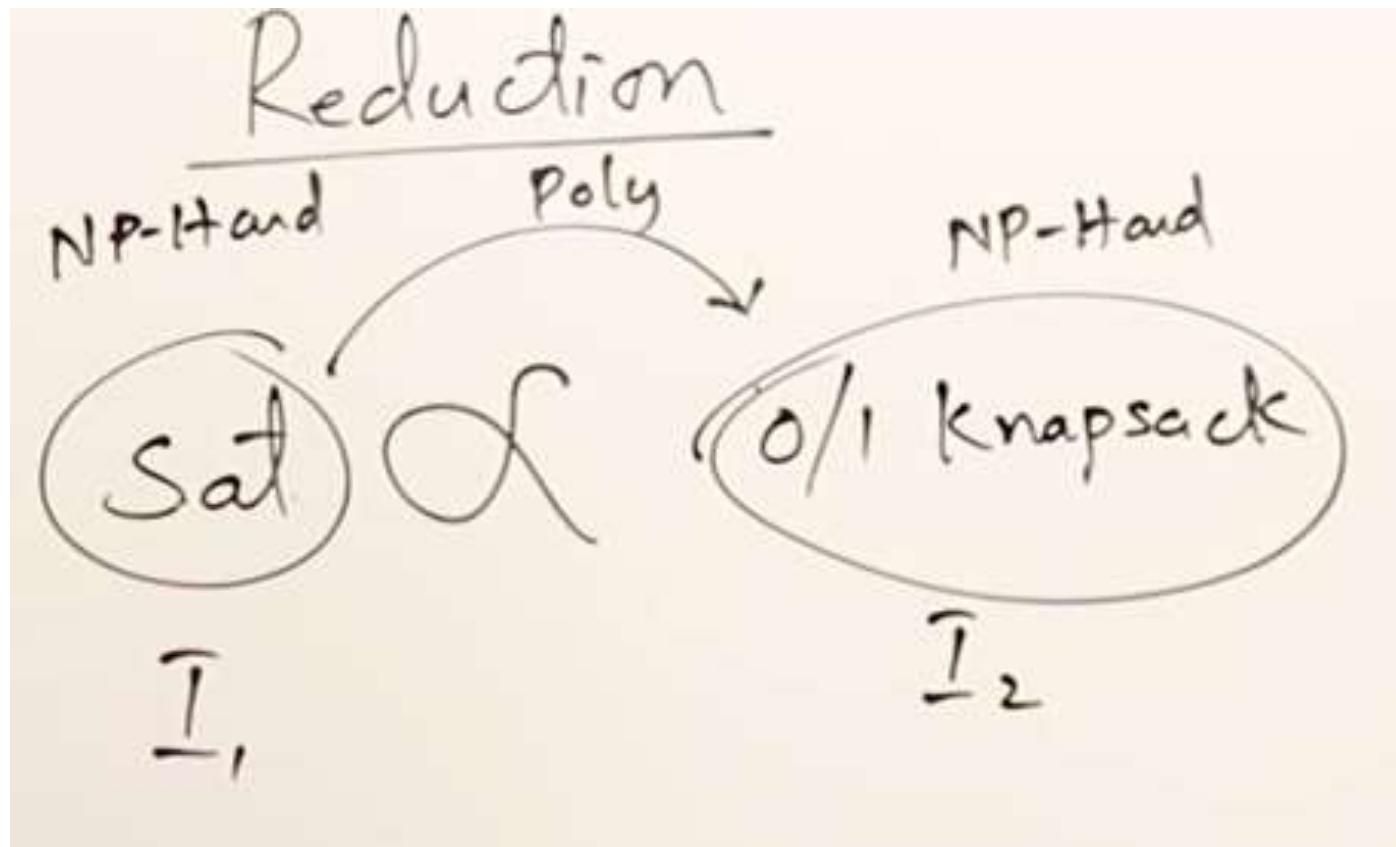
All the problems solved in exponential time complexity are hard problems. **Satisfiability problem is the base problem of all these problems.** We can call **satisfiability problem** as hard problem, NP- hard problem. If

satisfiability can solve in polynomial time all the other hard problems can also solve in polynomial time. In order to show the relationship between **satisfiability** and all the above problems we can use **the procedure reduction**

### **Polynomial-time reducibility**

Let L1 and L2 be problems. Problem L1 reduces to L2 (also written as  $L1 \alpha L2$ ) if and only if there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time.

For example: We take the example of satisfiability problem and we convert that problem into 0/1 knapsack problem and we say that if this 0/1 knapsack problem solved in polynomial time then the same algorithm can be used for solving satisfiability problem also in polynomial time and vice versa. They are related each other. Conversion is done in polynomial time. Here satisfiability is NP-hard so 0/1 Knapsack problem is also NP-hard.



If satisfiability problem can be reduced to L then L is also NP-Hard.  
Reduction has transitive property.

$SAT \alpha L_1, L_1 \alpha L_2 \rightarrow SAT \alpha L_2$

Non deterministic polynomial time algorithm for Satisfiability problem is existing so it is NP-Complete also. When we write a non-deterministic

polynomial time algorithm for NP-hard problem then it is in NP-Complete class.

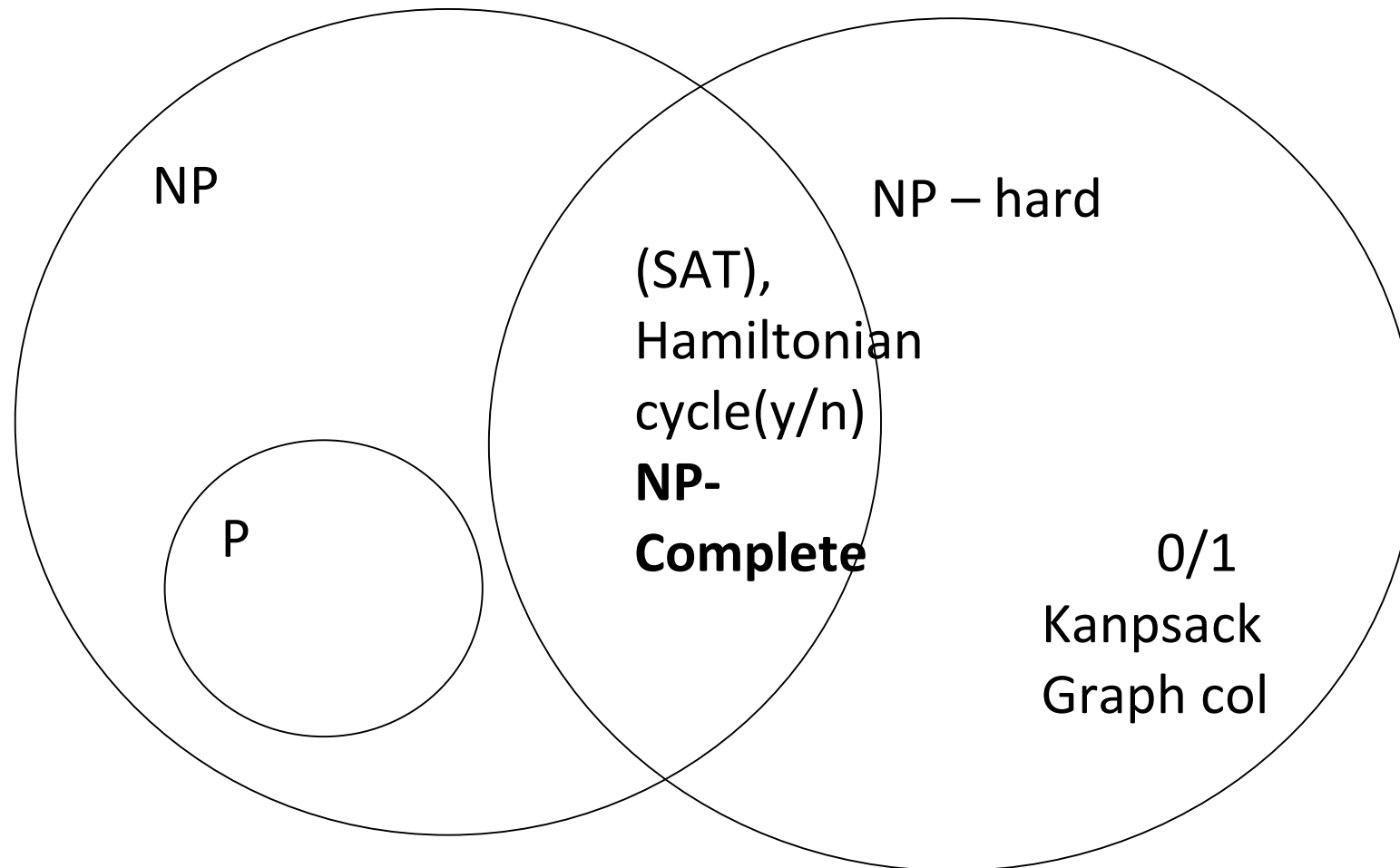
## **NP-complete problems**

A decision problem E is NP-complete if every problem in the class NP is polynomial-time reducible to E. The Hamiltonian cycle problem, the decision versions of the TSP and the graph coloring problem, as well as literally hundreds of other problems are known to be NP-complete.

## **NP-hard problems**

Optimization problems whose decision versions are NP complete are called NP-hard.

Following figure shows the relationship among P, NP, NP-complete, and NP-hard.

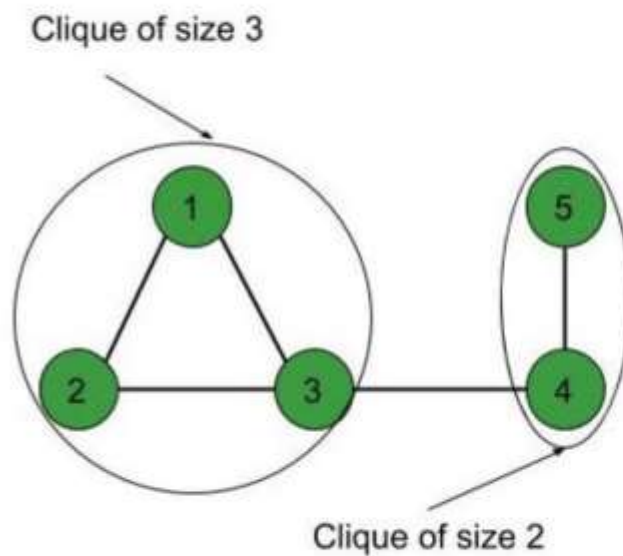


## **Clique Decision Problem (CDP)**

A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other that is the subgraph is a complete graph.

The Maximal Clique Problem is to find the maximum sized clique of a given graph  $G$ , that is a complete graph which is a subgraph of  $G$  and contains the maximum number of vertices. This is an optimization problem. Correspondingly, the Clique Decision Problem is to find if a clique of size  $k$  exists in the given graph or not.

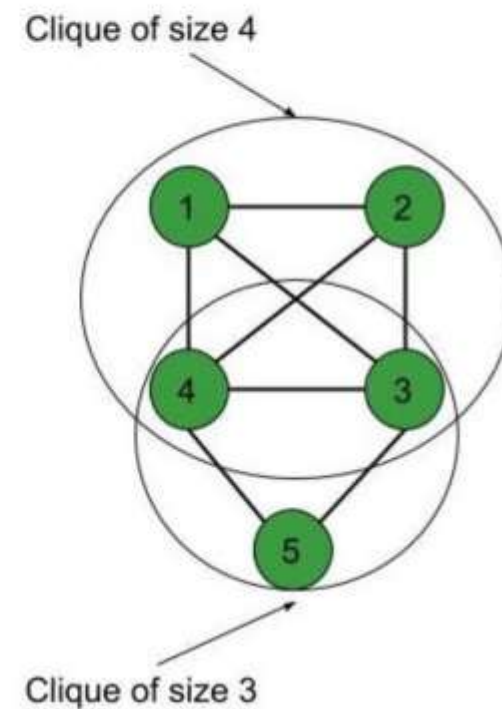




The above graph contains a maximum clique of size 3

Fig. (1)

\* A clique of size 2 is also present in Fig. (2)



The above graph contains a maximum clique of size 4

Fig. (2)



To prove that a problem is NP-Complete, we have to show that it belongs

to both NP and NP-Hard Classes. (Since NP-Complete problems are NP-Hard problems which also belong to NP)

### **The Clique Decision Problem belongs to NP**

If a problem belongs to the NP class, then it should have polynomial-time verifiability, that is given a certificate, we should be able to verify in polynomial time if it is a solution to the problem.

#### **Proof:**

1. Certificate – Let the certificate be a set  $S$  consisting of nodes in the clique and  $S$  is a subgraph of  $G$ .  $\{1,2,3\} \rightarrow S=k$
2. Verification – We have to check if there exists a clique of size  $k$  in the graph. Hence, verifying if number of nodes in  $S$  equals  $k$ , takes  $O(1)$  time. Verifying whether each vertex has an out-degree of  $(k-1)$  takes

$O(k^2)$  time. (Since in a complete graph, each vertex is connected to every other vertex through an edge. Hence the total number of edges in a complete graph  $= {}^kC_2 = \mathbf{k*(k-1)/2}$ ). Therefore, to check if the graph formed by the  $k$  nodes in  $S$  is complete or not, it takes  $O(k^2) = O(n^2)$  time (since  $k \leq n$ , where  $n$  is number of vertices in  $G$ ).

Therefore, the Clique Decision Problem has polynomial time verifiability and hence belongs to the NP Class.

### **The Clique Decision Problem belongs to NP-Hard**

A problem  $L$  belongs to NP-Hard if every NP problem is reducible to  $L$  in polynomial time. Now, let the Clique Decision Problem by  $C$ . To prove that  $C$  is NP-Hard, we take an already known NP-Hard problem, say  $S$ , and reduce it to  $C$  for a particular instance. If this reduction can be done

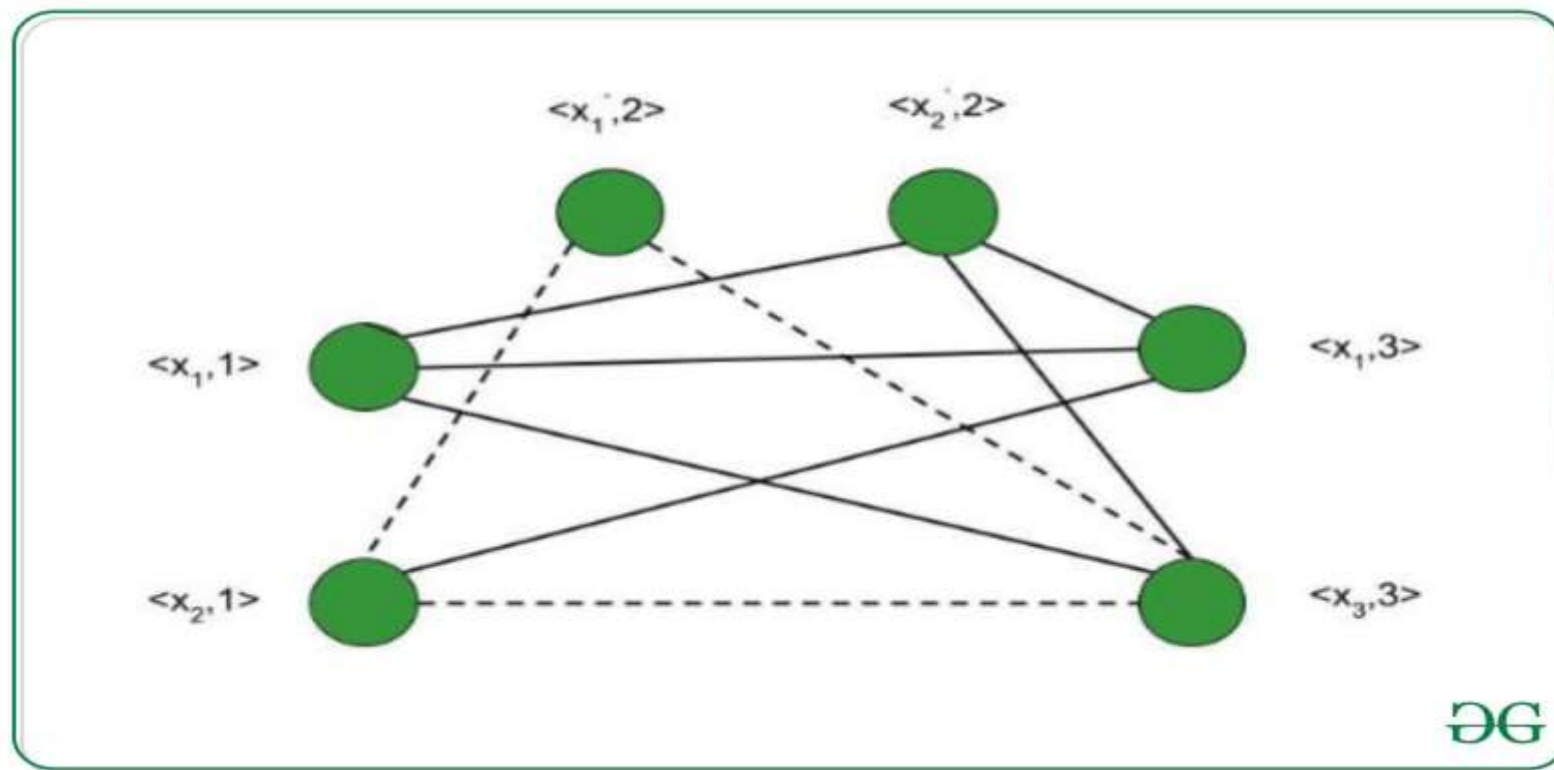
in polynomial time, then C is also an NP-Hard problem. The Boolean Satisfiability Problem (S) is an NP-Complete problem as proved by the Cook's theorem. Therefore, every problem in NP can be reduced to S in polynomial time. Thus, if S is reducible to C in polynomial time, every NP problem can be reduced to C in polynomial time, thereby proving C to be NP-Hard.

### **Proof that the Boolean Satisfiability problem reduces to the Clique Decision Problem**

Let the Boolean expression be –  $F = (x_1 \vee x_2) \wedge (x_1' \vee x_2') \wedge (x_1 \vee x_3)$  where  $x_1, x_2, x_3$  are the variables, ' $\wedge$ ' denotes logical 'and', ' $\vee$ ' denotes logical 'or' and  $x'$  denotes the **complement of x**. Let the expression within each parentheses be a clause. Hence we have three clauses –  $C_1, C_2$  and  $C_3$ .

Consider the vertices as –  $\langle x_1, 1 \rangle$ ;  $\langle x_2, 1 \rangle$ ;  $\langle x_1', 2 \rangle$ ;  $\langle x_2', 2 \rangle$ ;  $\langle x_1, 3 \rangle$ ;  $\langle x_3, 3 \rangle$  where the second term in each vertex denotes the clause number they belong to. We connect these vertices such that –

1. No two vertices belonging to the same clause are connected.
2. No variable is connected to its complement.



Thus, the graph  $G(V, E)$  is constructed such that –  $V = \{ \langle a, i \rangle \mid a \text{ belongs to } C_i \}$  and  $E = \{ ( \langle a, i \rangle, \langle b, j \rangle ) \mid i \text{ is not equal to } j ; b \text{ is not equal to } a' \}$ . Consider the subgraph of  $G$  with the vertices  $\langle x_2, 1 \rangle; \langle x_1', 2 \rangle; \langle x_3, 3 \rangle$ . It forms a clique of size 3 (Depicted by dotted line in above figure) . Corresponding to this, for the assignment –  $\langle x_1, x_2, x_3 \rangle = \langle 0, 1, 1 \rangle$  F

evaluates to true. Therefore, if we have  $k$  clauses in our satisfiability expression, we get a max clique of size  $k$  and for the corresponding assignment of values, the satisfiability expression evaluates to true. Hence, for a particular instance, the satisfiability problem is reduced to the clique decision problem.

Therefore, the **Clique Decision Problem is NP-Hard.**

## **Conclusion**

The Clique Decision Problem is NP and NP-Hard. Therefore, the Clique decision problem is **NP-Complete.**

## **Vertex Cover Problem (Node Cover Decision Problem (NCDP))**

The vertex-cover problem is to find a vertex cover of minimum size in a given undirected graph.

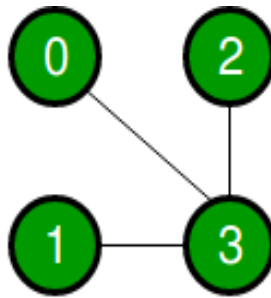
Vertex cover of a graph is a subset of vertices which cover every edge.

An edge is covered if one of its endpoint is chosen.

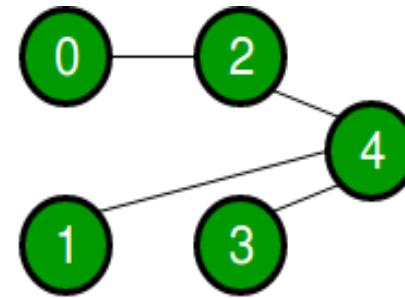
Eg.



Minimum vertex cover is  $\text{empty}\{\}$



Minimum vertex cover is  $\{3\}$



Minimum vertex cover is  $\{4, 2\}$  or  $\{4, 0\}$

It is solved by

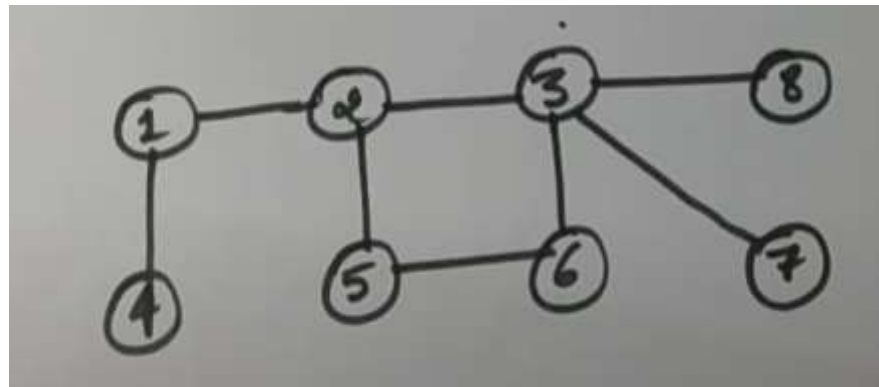
- Greedy Approach –Get best solution
- Approximation Method – Get nearby solution

Greedy method steps

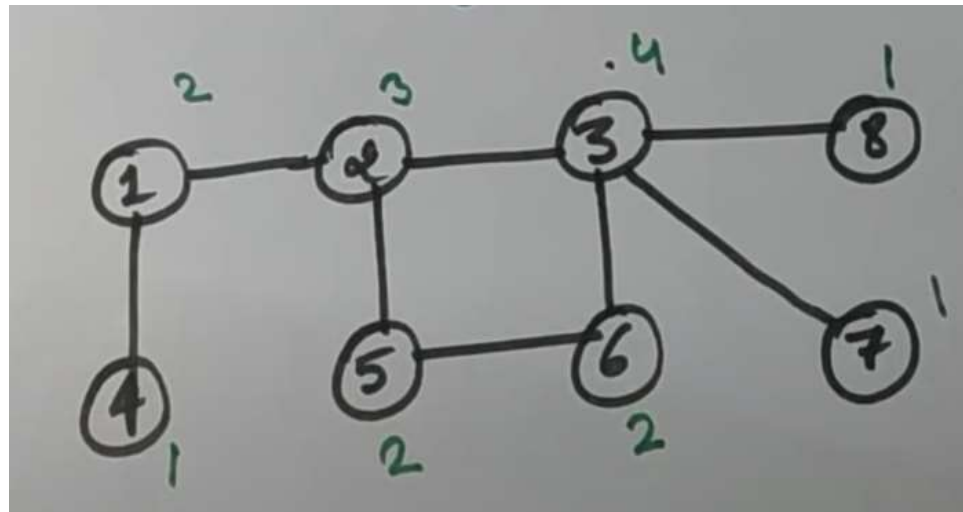


- Find a vertex  $v$  with maximum degree (Maximum number of edges associated with a vertex)
- Add  $v$  to the solution set and remove  $v$  and all its incident edges from the graph
- Repeat until all edges are covered.

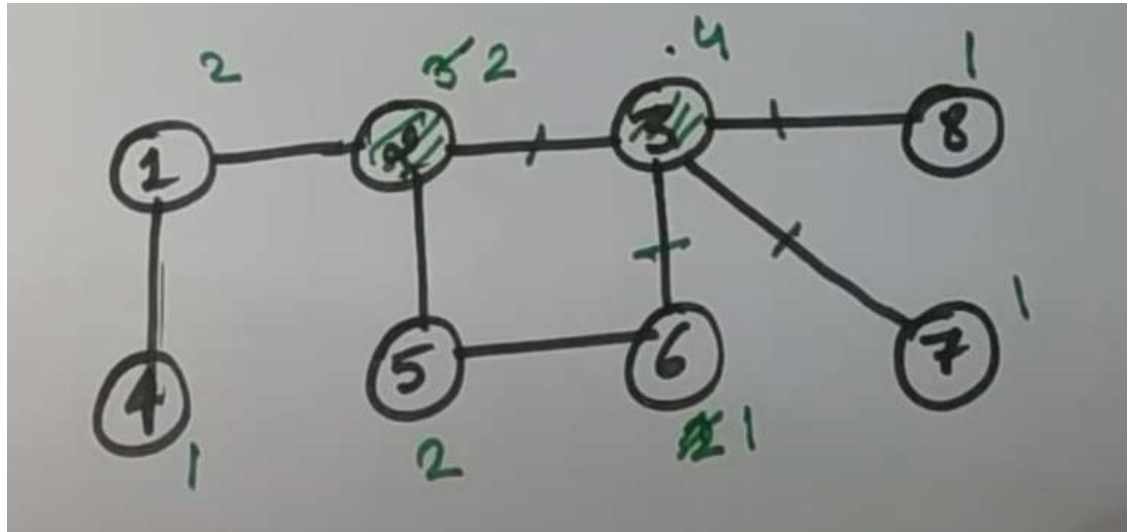
As an example, consider the graph



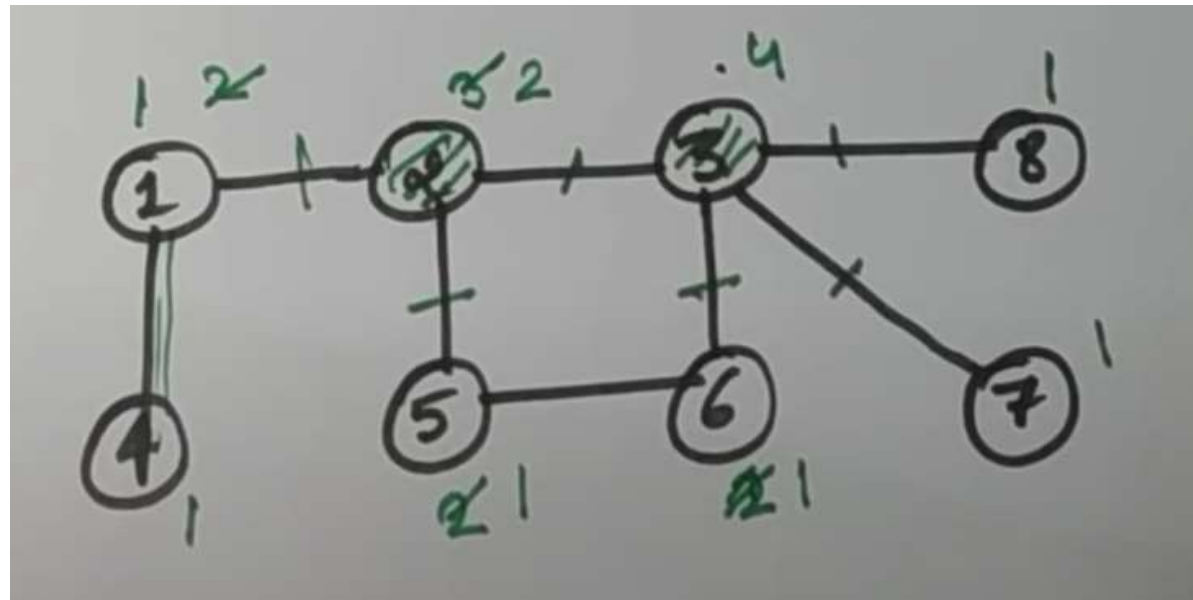
- Find the degree of all graph



- Choose the vertex which has maximum degree, vertex 3 and remove all the edges associated with vertex 3. Recalculate the degree of remaining vertices.
- Removed vertex set -  $\{3\}$



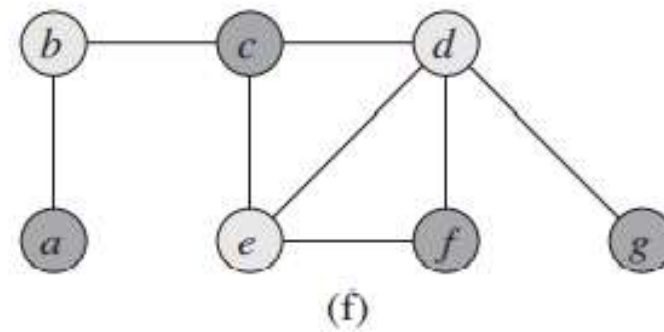
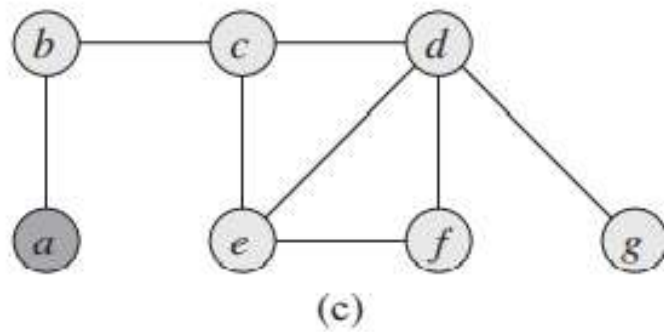
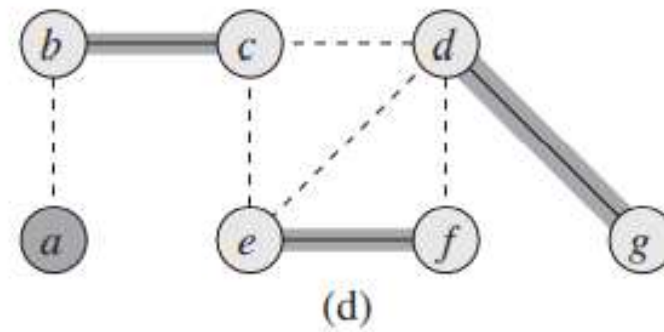
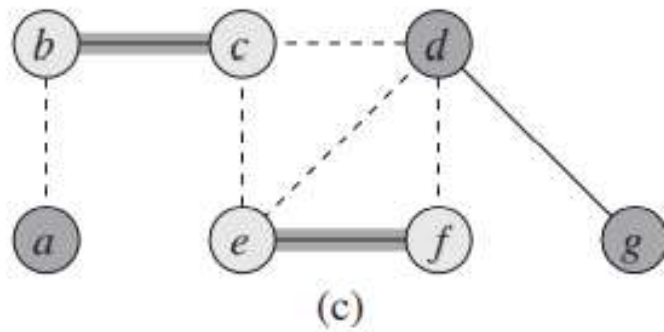
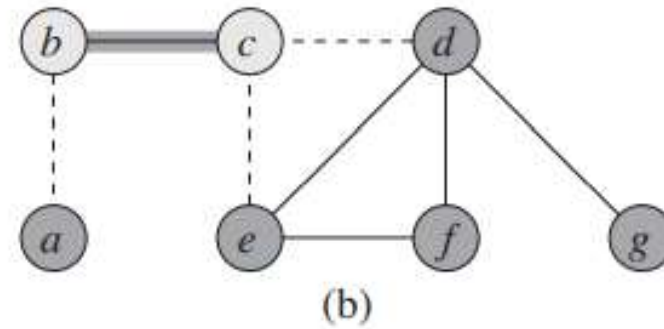
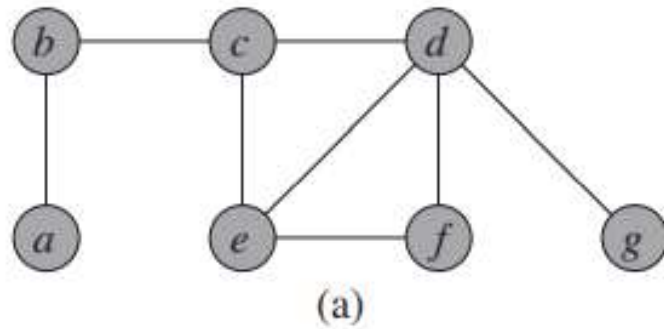
- Remove next vertex with minimum degree , vertex 2 and associated edges
- Removed vertex set -  $\{3, 2\}$
- Recalculate the degree and remove the vertex with minimum degree



- Removed vertex set -  $\{3, 2, 1\}$
- Next removed vertex is 5 so  $\{3, 2, 1, 5\}$  is new set
- $\{3, 5, 1\}$  and  $\{3, 1, 5\}$  are other solutions from this set. Time complexity is  $O(V+E)$
- It is not an optimal solution

## **APPROX Vertex Cover**

It is an optimal vertex cover. This problem is the optimization version of an NP-complete decision problem. Even though we don't know how to find an optimal vertex cover in a graph  $G$  in polynomial time, we can efficiently find a vertex cover that is near-optimal. The following approximation algorithm takes as input an undirected graph  $G$  and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.



## The operation of APPROX-VERTEX-COVER.

- (a) The input graph  $G$ , which has 7 vertices and 8 edges.
- (b) The edge  $(b, c)$  shown heavy, is the first edge chosen by APPROX-VERTEX COVER. Vertices  $b$  and  $c$ , shown lightly shaded, are added to the set  $C$  containing the vertex cover being created. Edges  $(a, b)$ ,  $(c, e)$ , and  $(c, d)$  shown dashed, are removed since they are now covered by some vertex in  $C$ .
- (c) Edge  $(e, f)$  is chosen; vertices  $e$  and  $f$  are added to  $C$ .
- (d) Edge  $(d, g)$  is chosen; vertices  $d$  and  $g$  are added to  $C$ .
- (e) The set  $C$ , which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices  $b, c, d, e, f, g$ .
- (f) The optimal vertex cover for this problem contains only three vertices:  **$b, d$ , and  $e$ .**

## Algorithm

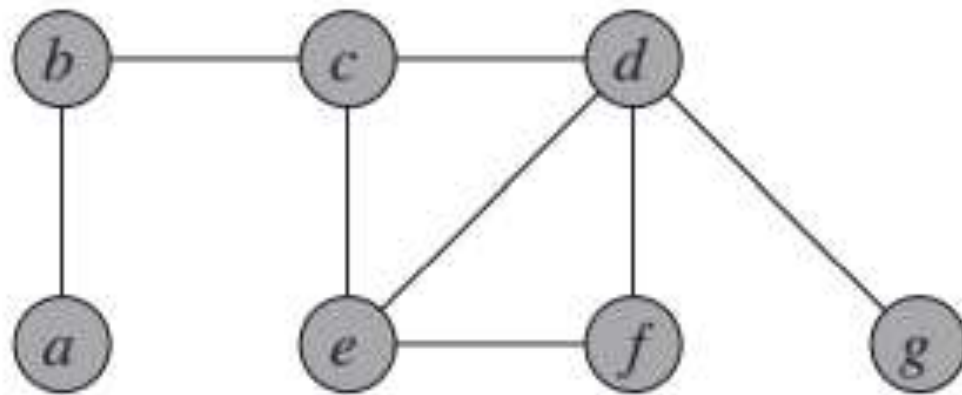
APPROX-VERTEX-COVER( $G$ )

```
1   $C = \emptyset$ 
2   $E' = G.E$ 
3  while  $E' \neq \emptyset$ 
4      let  $(u, v)$  be an arbitrary edge of  $E'$ 
5       $C = C \cup \{u, v\}$ 
6      remove from  $E'$  every edge incident on either  $u$  or  $v$ 
7  return  $C$ 
```

The running time of this algorithm is  $O(V + E)$ , using adjacency lists to represent  $E'$



## Optimal solution



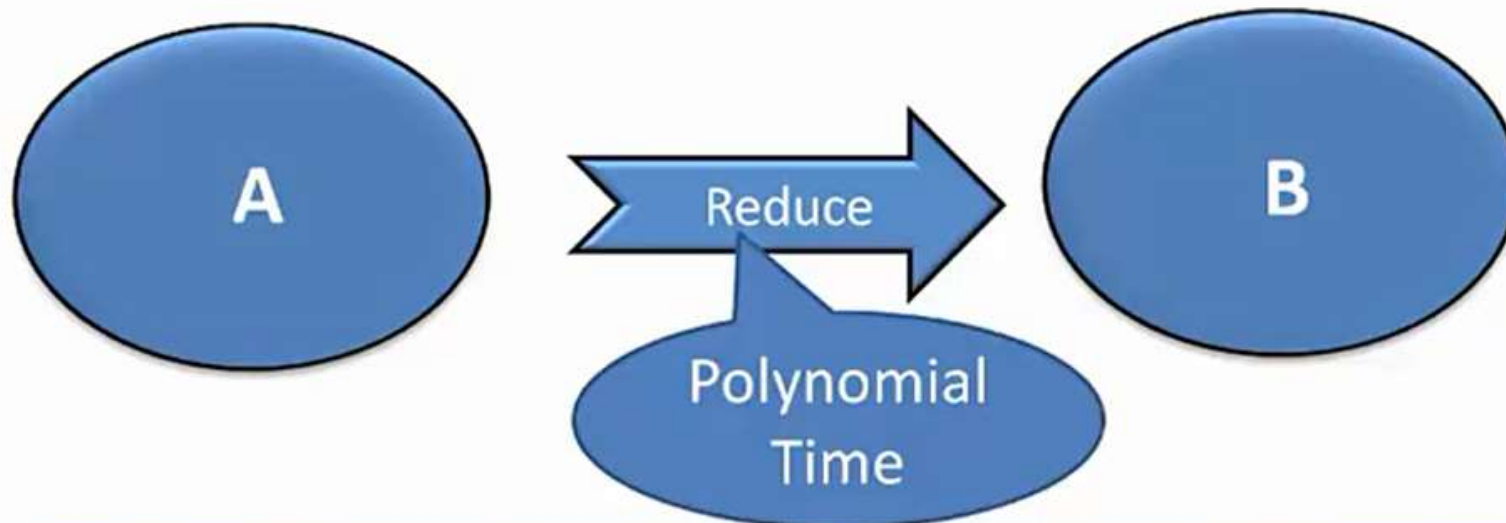
Select the vertex which has highest degree. That is node d.

Remove all the edges connected into it.

Select the highest degree vertex in the rest of the vertices (a, b, c, e and f), that is b or c or e. Select b, remove the edges, after that choose e and remove edge. Final list of  $C = \{b, d, e\}$

# NP-Completeness of Vertex Cover

Reduction:



Let A and B are two problems then problem A reduces to problem B iff there is a way to solve A by deterministic algorithm that solve B in polynomial time.

If A is reducible to B we denote it by  $A \propto B$

### Properties:

1. if A is reducible to B and B in P then A in P.
2. A is not in P implies B is not in P

If A and B are two languages. If A is reducible to B in polynomial time then B is NP-Hard.

If B is in NP, then B is NP-Complete.

Steps for proving NP-Complete:

Step 1: Prove that B is in NP

Step 2: Select an NP-Complete Language A.

Step 3: Construct a function  $f$  that maps members of A to members of B.

Step 4: Show that  $x$  is in A iff  $f(x)$  is in B.

Step 5: Show that  $f$  can be computed in polynomial time.

## NP-Completeness of Vertex Cover Problem:

To Show Vertex Cover Problem is in NP.

A Problem which cannot be solved on polynomial time but is verified in polynomial time is known as Non Deterministic Polynomial or NP-Class Problem.

Given  $V_c$ , vertex cover of  $G = (V, E)$ ,  $|V_c| = k$ . We can check in  $O(|V| + |E|)$  that  $V_c$  is a vertex cover for  $G$ .

For each vertex  $\in V_c$ , remove all incident edges.  
Check if all edges were removed from  $G$ .

Thus Vertex Cover Problem is in NP.



NP-Completeness of Vertex Cover Problem:

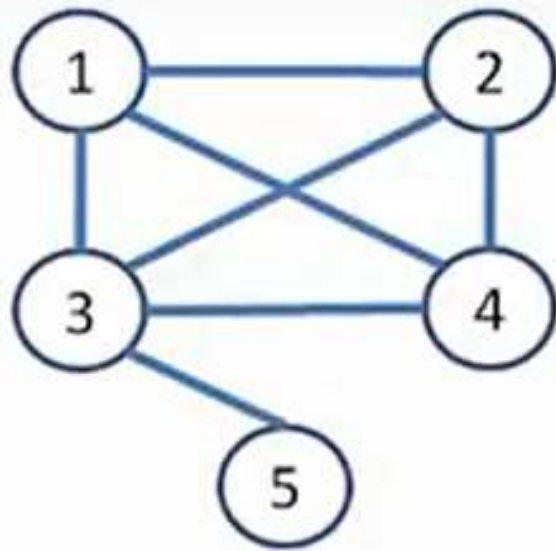
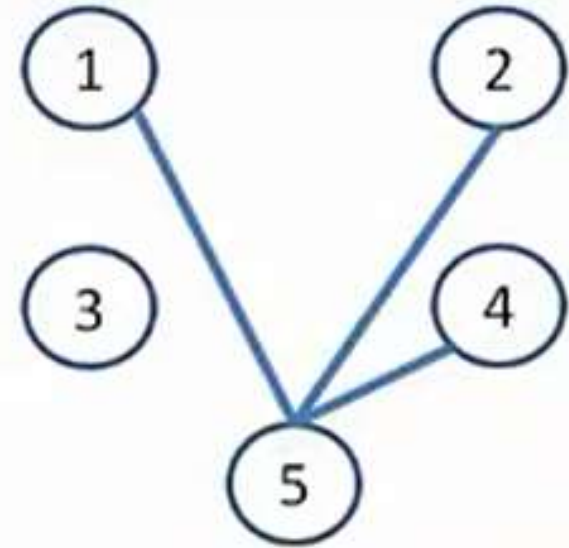
To show Vertex Cover Problem is NP- Hard.

we need to show that Vertex Cover is at least as hard any other problem in NP.

we give a reduction from Clique to Vertex Cover Problem.

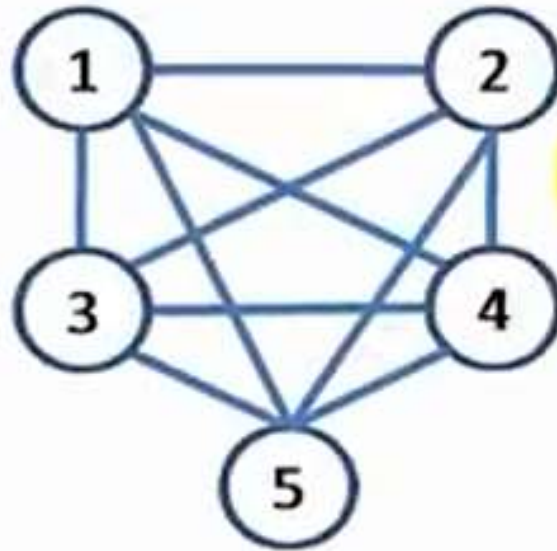
It means, given an instance  $I$  of Clique, we will produce a graph  $G(V,E)$  and an integer  $k$  such that  $G$  has a maximum clique of  $k$  if and only if  $I$  in  $\bar{G}(V,\bar{E})$  has a vertex cover of size  $|V|-k$ .

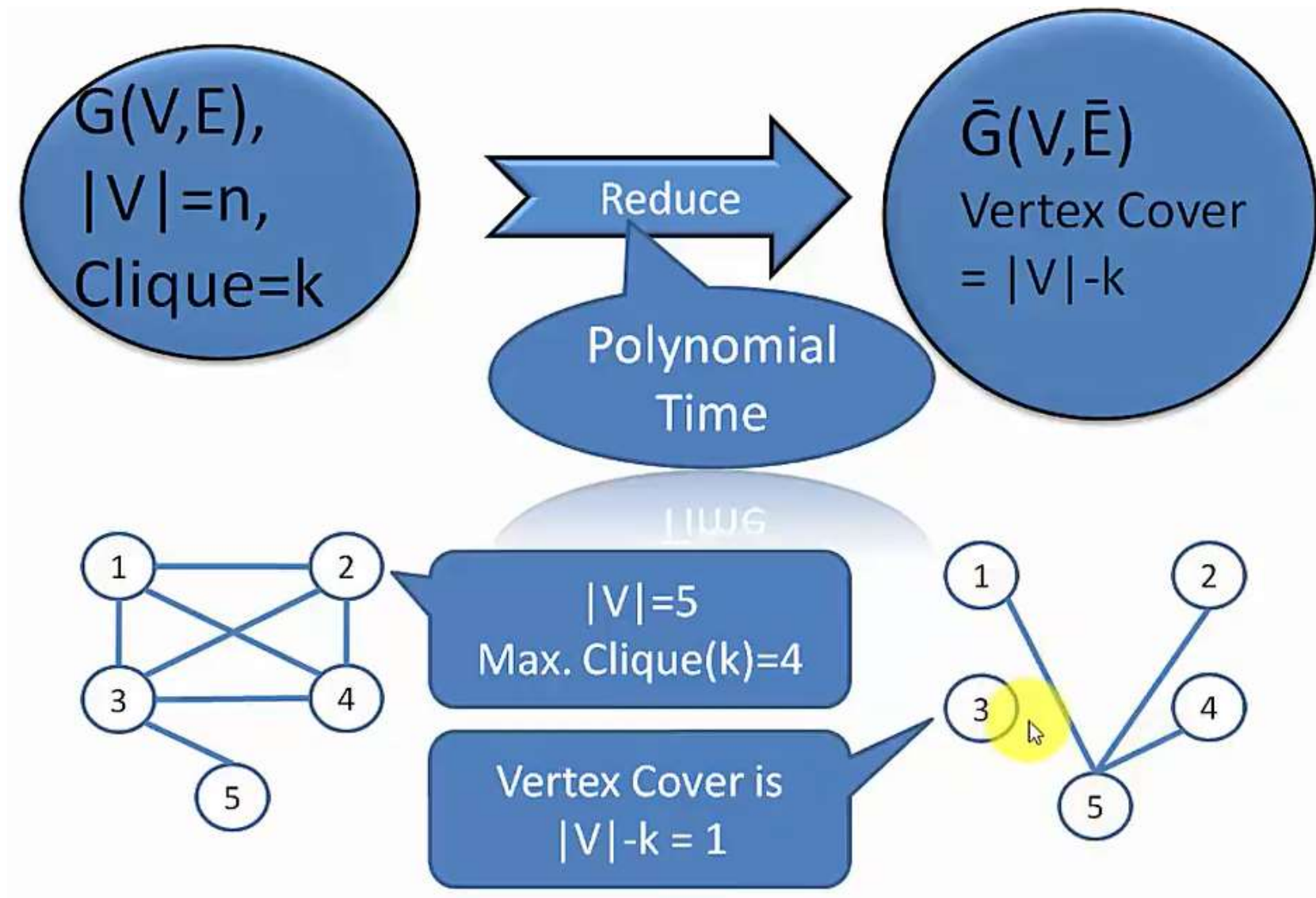
$G'$  is the complement of  $G$

 $G(V,E)$  $\bar{G}(V,\bar{E})$ 

When the above graphs are merged it will become a complete graph.



$G(V,E)$  $\bar{G}(V,\bar{E})$ 



Let  $G$  has clique  $V'$  of size  $k$ .

$\Rightarrow G$  has vertex cover of size  $|V| - k$

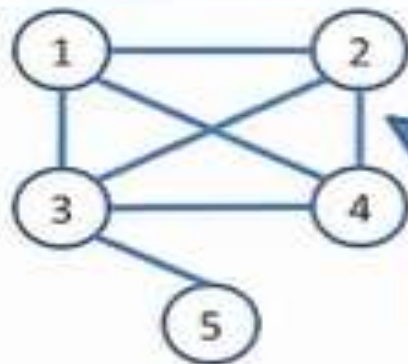
$(a, b) \in E \Rightarrow (a, b) \notin \bar{E}$

If  $(a, b) \in \bar{E}$ , then at least  $a$  or  $b \notin V'$ .

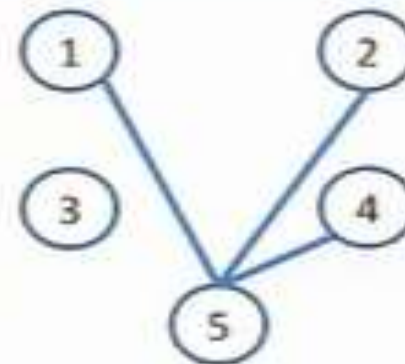
Every pair in  $V'$  is connected by an edge in  $E$ .

$\Rightarrow$  At least one of  $a$  or  $b$  is in  $V - V'$

$\Rightarrow$  Edge  $(a, b)$  is covered by  $V - V'$



$V = \{1, 2, 3, 4, 5\}$   
 $V' = \{1, 2, 3, 4\}$   
 $V - V' = \{5\}$



For example vertex  $(a,b)=(1,3)$  in  $E$  but vertex  $(1,3)$  not belongs to  $E'$ .

If  $(a,b)=(1,5)$  belongs to  $E'$ , but  $(1,5)$  not belongs to  $E$ .

But at least one  $a$  or  $b$  is in  $V-V'$ .

## Flow Networks and Flows

### Flow networks and flows

---

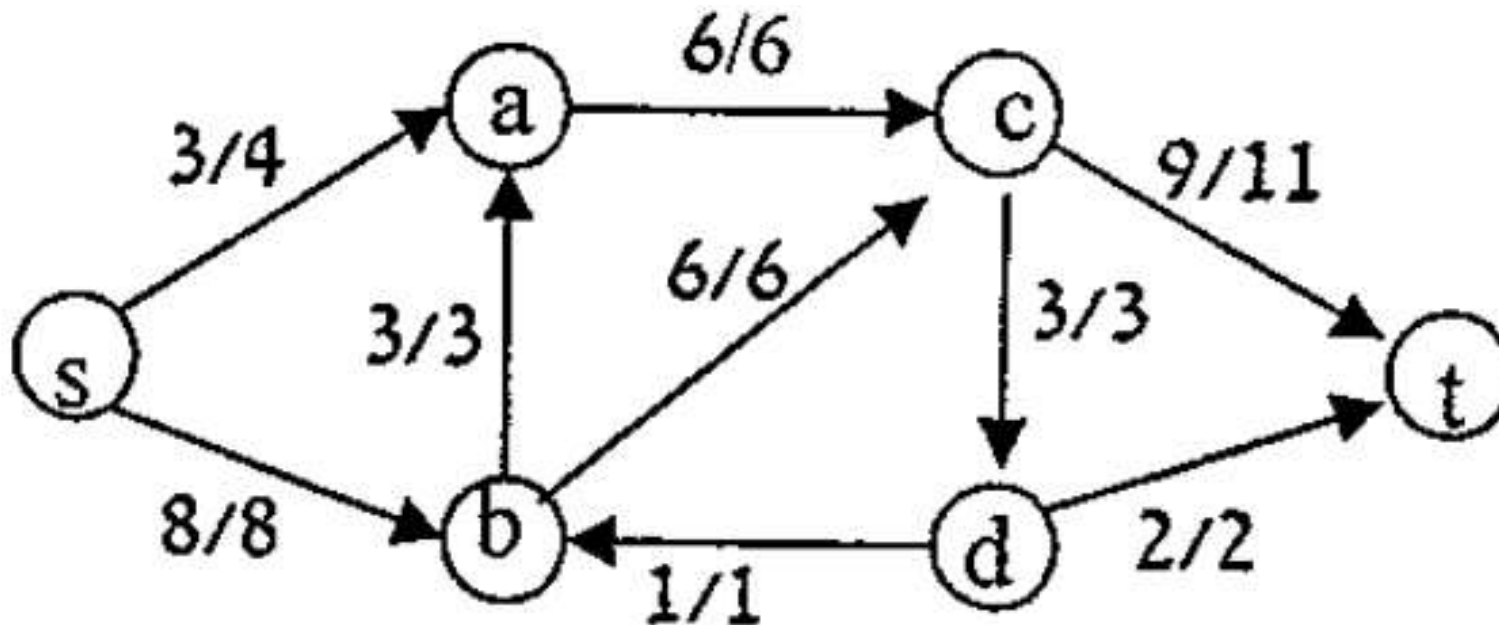
A *flow network*  $G = (V, E)$  is a directed graph in which each edge  $(u, v) \in E$  has a nonnegative *capacity*  $c(u, v) \geq 0$ . We further require that if  $E$  contains an edge  $(u, v)$ , then there is no edge  $(v, u)$  in the reverse direction. (We shall see shortly how to work around this restriction.) If  $(u, v) \notin E$ , then for convenience we define  $c(u, v) = 0$ , and we disallow self-loops. We distinguish two vertices in a flow network: a *source*  $s$  and a *sink*  $t$ . For convenience, we assume that each vertex lies on some path from the source to the sink. That is, for each vertex  $v \in V$ , the flow network contains a path  $s \rightsquigarrow v \rightsquigarrow t$ . The graph is therefore connected and, since each vertex other than  $s$  has at least one entering edge,  $|E| \geq |V| - 1$ .

Flow Network is a directed graph that is used for modelling material Flow. There are two different vertices; one is a **source** which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed. The flow of the material at any mark in the system is the rate at which the element moves.

Some real-life problems like the flow of liquids through pipes, the current through wires and delivery of goods can be modelled using flow networks.

**Definition:** A Flow Network is a directed graph  $G = (V, E)$  such that

1. For each edge  $(u, v) \in E$ , we associate a nonnegative weight capacity  $c(u, v) \geq 0$ . If  $(u, v) \notin E$ , we assume that  $c(u, v) = 0$ .
2. There are two distinguishing points, the source  $s$ , and the sink  $t$ ;
3. For every vertex  $v \in V$ , there is a path from  $s$  to  $t$  containing  $v$ .



Let  $G = (V, E)$  be a flow network. Let  $s$  be the source of the network, and let  $t$  be the sink. A flow in  $G$  is a real-valued function  $f: V \times V \rightarrow \mathbb{R}$  such that the following properties hold:

- **Capacity Constraint:** For all  $u, v \in V$ , we need  $f(u, v) \leq c(u, v)$ . (Each edge  $(u, v)$  have a capacity given as  $c(u, v)$ ).
- **Skew Symmetry:** For all  $u, v \in V$ , we need  $f(u, v) = -f(v, u)$ . (+ value shows the incoming-in and – value shows the outgoing-out)

- **Flow Conservation:** For all  $u \in V - \{s, t\}$ , (nodes except  $s$  and  $t$ ) we need

$$\sum_{v \in V} f(u, v) = \sum_{u \in V} f(u, v) = 0$$

(+ value shows the incoming-in and – value shows the outgoing-out – from the figure  $s$  to  $a = 3$  +  $s$  to  $b = 8$ ,  **$3+8=11$** ,  $c$  to  $t = 9$  +  $c$  to  $d = 2$ ,  **$9+2=11$** )

(  **$+3+8$  and  $-9+2=0$** )

The quantity  $f(u, v)$ , which can be positive or negative, is known as the net flow from vertex  $u$  to vertex  $v$ . In the **maximum-flow problem**, we are given a flow network  $G$  with source  $s$  and sink  $t$ , and we wish to find a flow of maximum value from  $s$  to  $t$ .

The three properties can be described as follows:

1. **Capacity Constraint** makes sure that the flow through each edge is not greater than the capacity.

**2.Skew Symmetry** means that the flow from  $u$  to  $v$  is the negative of the flow from  $v$  to  $u$ .

**3.The flow-conservation** property says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of flow into a  $v$  is the same as the amount of flow out of  $v$  for every vertex  $v \in V - \{s, t\}$

The value of the flow is the net flow from the source,

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

The **positive net flow entering** a vertex  $v$  is described by

$$\sum_{\{u \in V : f(u, v) > 0\}} f(u, v)$$

The **positive net flow** leaving a vertex is described symmetrically. One interpretation of the Flow-Conservation Property is that the positive net



flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow  $f$  is said to be **integer-valued** if  $f(u, v)$  is an integer for all  $(u, v) \in E$ . Clearly, the value of the flow is an integer is an integer-valued flow.

## Ford Fulkerson method

The Ford-Fulkerson algorithm is used to **detect maximum flow from start vertex to sink vertex** in a given graph. In this graph, every edge has the capacity. Two vertices are provided named Source and Sink. The source vertex has all outward edge, no inward edge, and the sink will have all inward edge no outward edge. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems:

- Residual networks
- Augmenting paths
- Cuts

The Ford-Fulkerson method iteratively increases the value of the flow. We start with  $f(u, v) = 0$  for all  $u, v \in V$ , giving an initial flow of value 0. At each iteration, we increase the flow value in  $G$  by finding an “augmenting path” in an associated “residual network”  $G_f$ . Once we know the edges of an augmenting path in  $G_f$ , we can easily identify specific edges in  $G$  for which we can change the flow so that we increase the value of the flow. Although each iteration of the Ford-Fulkerson method increases the value of the flow, we shall see that the flow on any particular edge of  $G$  may increase or decrease; decreasing the flow on some edges may be necessary in order to enable an algorithm to send more flow from the source to the sink. We repeatedly augment the flow until the residual network has no more augmenting paths. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

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

- 1 initialize flow  $f$  to 0
- 2 **while** there exists an augmenting path  $p$  in the residual network  $G_f$
- 3     augment flow  $f$  along  $p$
- 4 **return**  $f$

In order to implement and analyze the Ford-Fulkerson method, we need to introduce several additional concepts.

**Residual networks**

Intuitively, given a flow network  $G$  and a flow  $f$ , the residual network  $G_f$  consists of edges with capacities that represent how we can change the flow on edges of  $G$ . An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. If that value is positive, we place that edge into  $G_f$  with a “residual capacity” of  $c_f(u, v) = c(u, v) - f(u, v)$ .



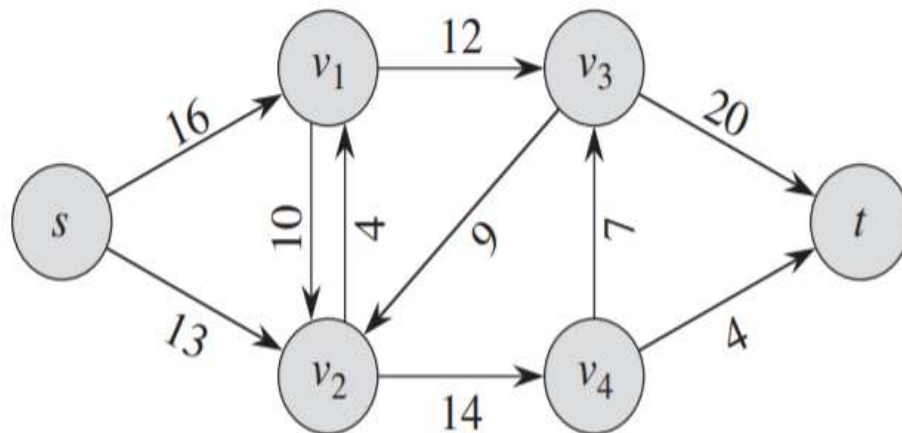
The only edges of  $G$  that are in  $G_f$  are those that can admit more flow; those edges  $(u, v)$  whose flow equals their capacity have  $c_f(u, v) = 0$ , and they are not in  $G_f$ .

The residual network  $G_f$  may also contain edges that are not in  $G$ , however. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge. In order to represent a possible decrease of a positive flow  $f(u, v)$  on an edge in  $G$ , we place an edge  $(v, u)$  into  $G_f$  with residual capacity  $c_f(v, u) = f(u, v)$ —that is, an edge that can admit flow in the opposite direction to  $(u, v)$ , at most canceling out the flow on  $(u, v)$ . These reverse edges in the residual network allow an algorithm to send back flow it has already sent along an edge. Sending flow back along an edge is equivalent to *decreasing* the flow on the edge, which is a necessary operation in many algorithms.

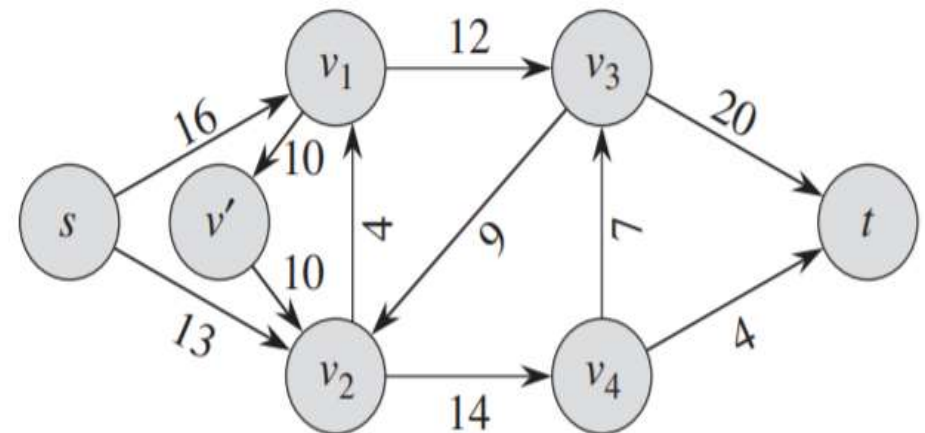
More formally, suppose that we have a flow network  $G = (V, E)$  with source  $s$  and sink  $t$ . Let  $f$  be a flow in  $G$ , and consider a pair of vertices  $u, v \in V$ . We define the **residual capacity**  $c_f(u, v)$  by

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E, \\ f(v, u) & \text{if } (v, u) \in E, \\ 0 & \text{otherwise.} \end{cases} \quad (26.2)$$

Because of our assumption that  $(u, v) \in E$  implies  $(v, u) \notin E$ , exactly one case in equation (26.2) applies to each ordered pair of vertices.



(a)



(b)

As an example of equation (26.2), if  $c(u, v) = 16$  and  $f(u, v) = 11$ , then we can increase  $f(u, v)$  by up to  $c_f(u, v) = 5$  units before we exceed the capacity constraint on edge  $(u, v)$ . We also wish to allow an algorithm to return up to 11 units of flow from  $v$  to  $u$ , and hence  $c_f(v, u) = 11$ .

Given a flow network  $G = (V, E)$  and a flow  $f$ , the **residual network** of  $G$  induced by  $f$  is  $G_f = (V, E_f)$ , where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} . \quad (26.3)$$

That is, as promised above, each edge of the residual network, or **residual edge**, can admit a flow that is greater than 0. ~~Figure 26.4(a) repeats the flow network  $G$~~

## Augmenting paths

Given a flow network  $G = (V, E)$  and a flow  $f$ , an *augmenting path*  $p$  is a simple path from  $s$  to  $t$  in the residual network  $G_f$ . By the definition of the residual network, we may increase the flow on an edge  $(u, v)$  of an augmenting path by up to  $c_f(u, v)$  without violating the capacity constraint on whichever of  $(u, v)$  and  $(v, u)$  is in the original flow network  $G$ .

~~The shaded path in Figure 26.4(b) is an augmenting path. Treating the residual network  $G_f$  in the figure as a flow network, we can increase the flow through each edge of this path by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is  $c_f(v_2, v_3) = 4$ . We call the maximum amount by which we can increase the flow on each edge in an augmenting path  $p$  the *residual capacity* of  $p$ , given by~~

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

Augmenting path: Augmenting path can be done in two ways:

1. Non-full forward edges
2. Non-empty backward edges.



## Minimal Cut

It is also known as bottle neck capacity which decides maximum flow from source to sink through an augmented path.

## Definition of a Cut

A cut (also called as st-cut) is the division of graph into two partitions A and B such that some nodes are in partition A and some nodes in partition B and should satisfy the constraint that s must belongs to A while t must belongs to B. The capacity of an s-t cut is defined by the sum of the capacity of each edge in the cut-set.

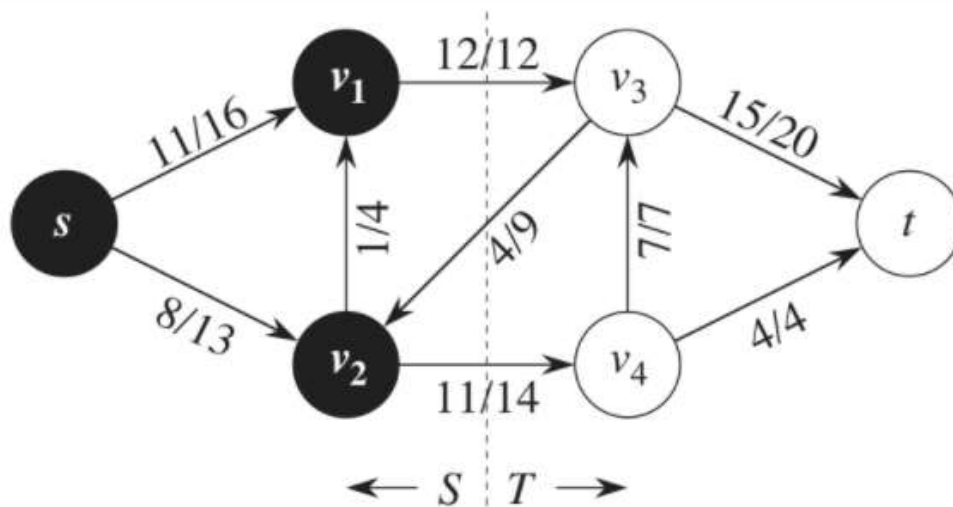
The max-flow min-cut theorem states that in a flow network, the amount of maximum flow is equal to capacity of the minimum cut.

A cut  $(S, T)$  of flow network  $G = (V, E)$  is a partition of  $V$  into  $S$  and

$T = V - S$  such that  $s \in S$  and  $t \in T$ . If  $f$  is a flow, then the net flow  $f(S, T)$  across the cut  $(S, T)$  is defined to be



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



**Figure 26.5** A cut  $(S, T)$  in the flow network of Figure 26.1(b), where  $S = \{s, v_1, v_2\}$  and  $T = \{v_3, v_4, t\}$ . The vertices in  $S$  are black, and the vertices in  $T$  are white. The net flow across  $(S, T)$  is  $f(S, T) = 19$ , and the capacity is  $c(S, T) = 26$ .

The *capacity* of the cut  $(S, T)$  is

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

A *minimum cut* of a network is a cut whose capacity is minimum over all cuts of the network.

The asymmetry between the definitions of flow and capacity of a cut is intentional and important. For capacity, we count only the capacities of edges going from  $S$  to  $T$ , ignoring edges in the reverse direction. For flow, we consider the flow going from  $S$  to  $T$  minus the flow going in the reverse direction from  $T$  to  $S$ .

The net flow across this network is

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) &= 12 + 11 - 4 \\ &= 19, \end{aligned}$$

and the capacity of this cut is

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26. \end{aligned}$$

## 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  $(S, T)$  of  $G$ .

We want to prove

$$1 \Leftrightarrow 2 \Leftrightarrow 3 \Leftrightarrow 1$$

In a network flow, maximum amount of flow passing from source to sink is equal to the minimum of the capacities of all possible cuts.

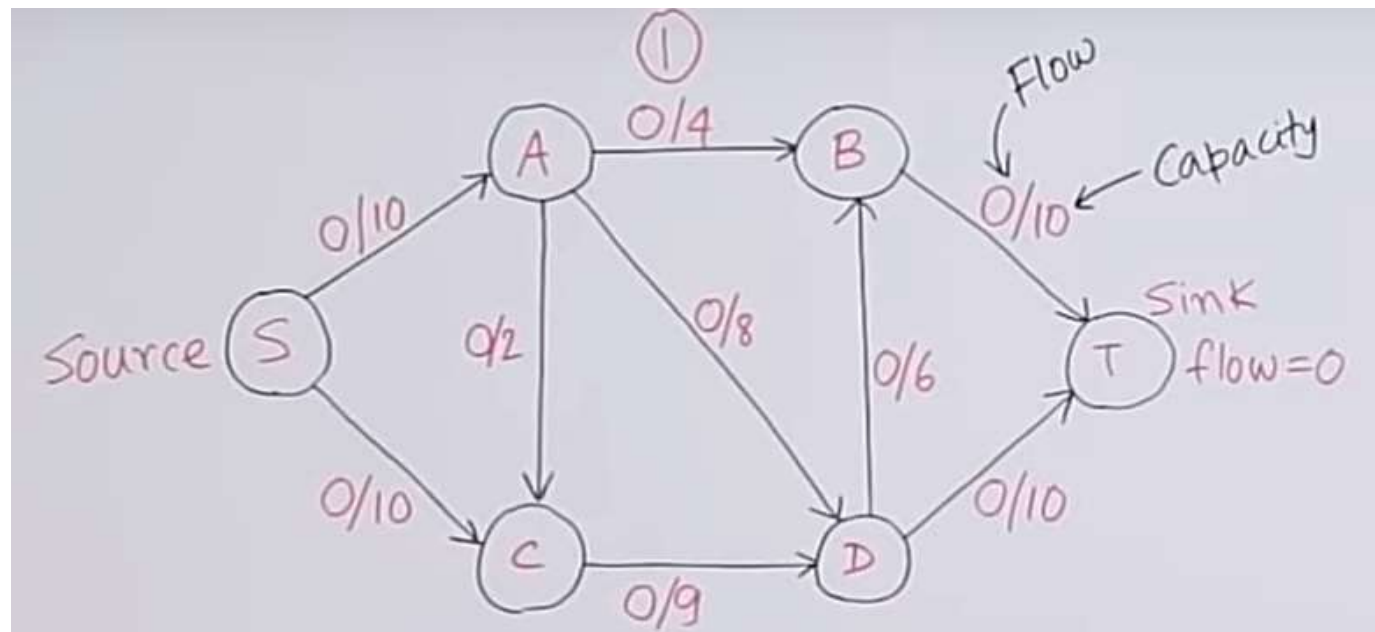
Proof: Let  $N$  be a given network and  $S$  be the source and  $T$  be the sink

Let  $S-T$  be any cut in the network, so in any  $S-T$  of the network there will be an edge from the path connecting source and sink

Thus every flow from source to sink must pass through one or more edges of  $S-T$ . Hence the total flow between source and sink can't exceed the minimum capacity of  $S-T$ .

Since this is true for every cut the flow can't exceed but can be equal to the minimum of the capacities of all the possible cuts. Hence the proof.

Explanation with an example



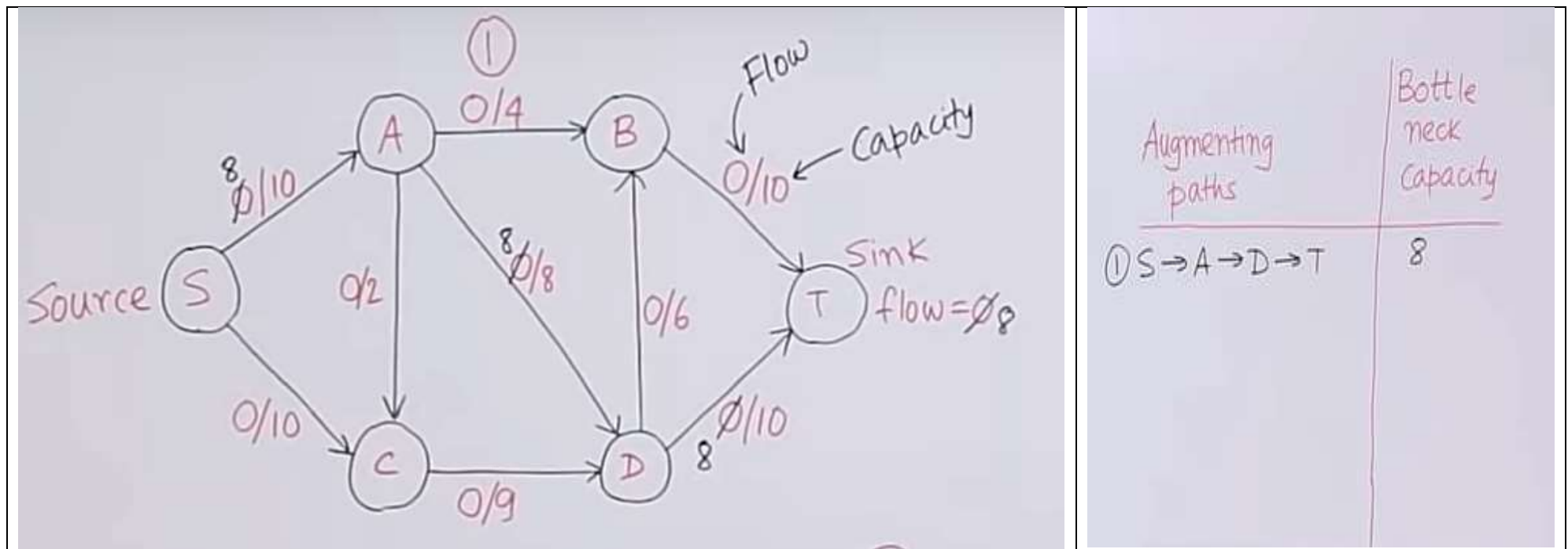
In the above figure we can see initially consider flow as 0.

Next when we find out an augmenting path then flow will calculate and add with main flow.

Select the first augmenting path as  $S \rightarrow A \rightarrow D \rightarrow T$ .

Bottle neck capacity is 8. (10, 8, 10  $\rightarrow$  minimum is 8).

Flow=0

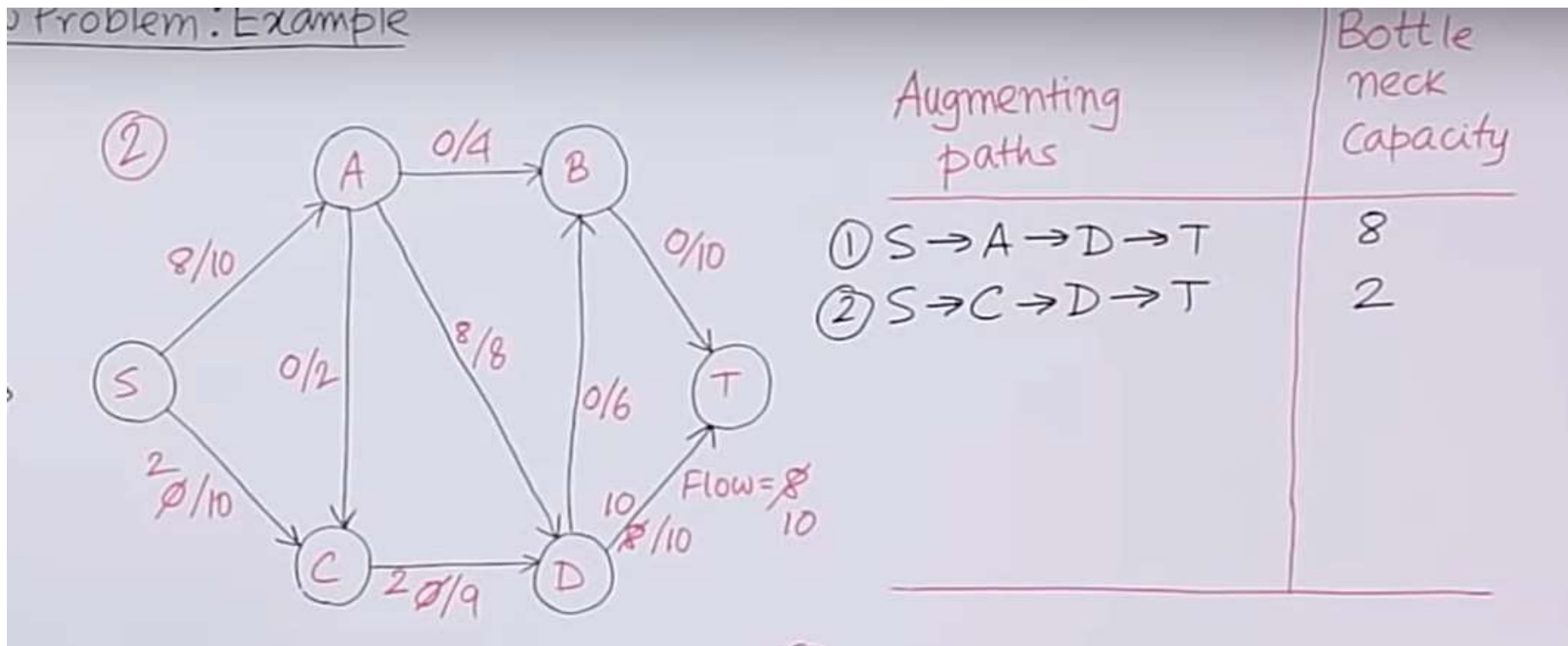


Now the Residual capacity is 2,0,2

Flow=8

Select the next path S->C->D->T

Bottle neck capacity = 2



Flow is  $8+2=10$

Next path we are selecting is  $S \rightarrow C \rightarrow D \rightarrow A \rightarrow B \rightarrow T$

Augmenting path: Augmenting path can be done in two ways:

3. Non-full forward edges
4. Non-empty backward edges.

Eg. of Non full forward edge

Edge SC has the capacity 10 only 2 is flowing, so not full

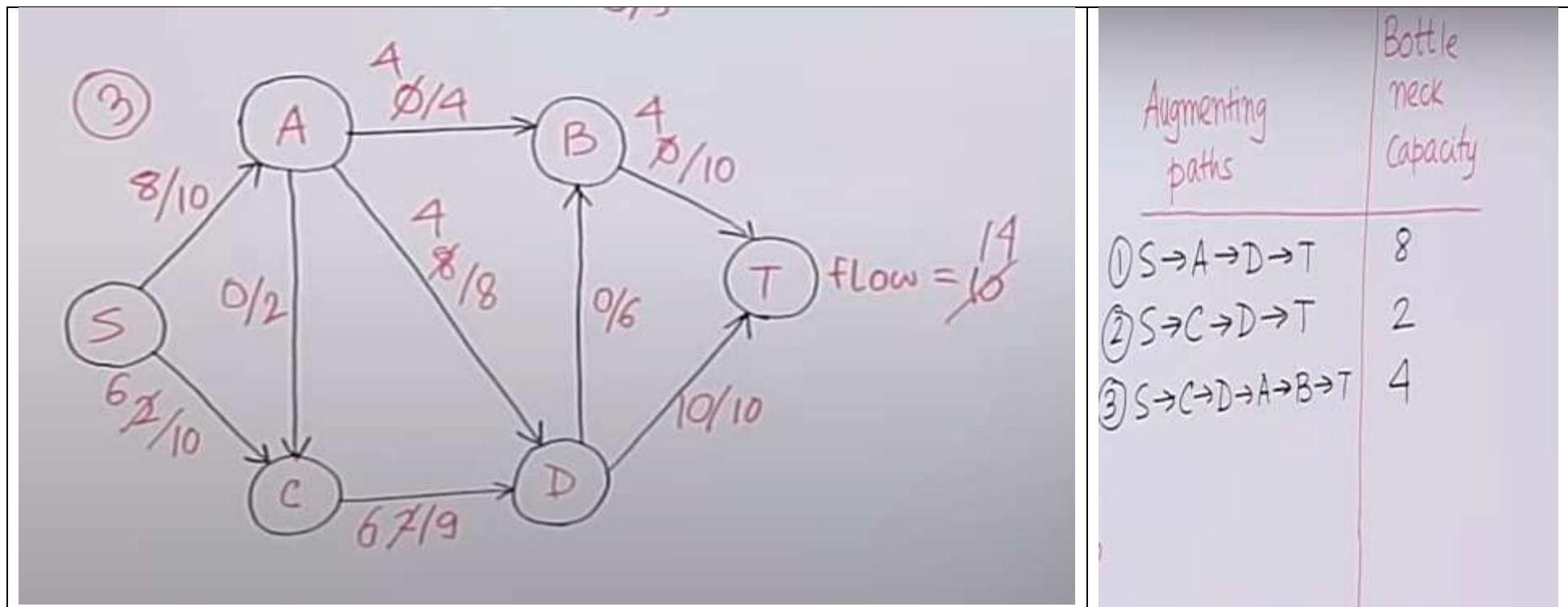
Eg. of Non-empty backward edges

DA is backward edge which is non zero. It has capacity 8.

Next case of bottle neck capacity  $\rightarrow$  (out of 10,9,8,4,10) – 4

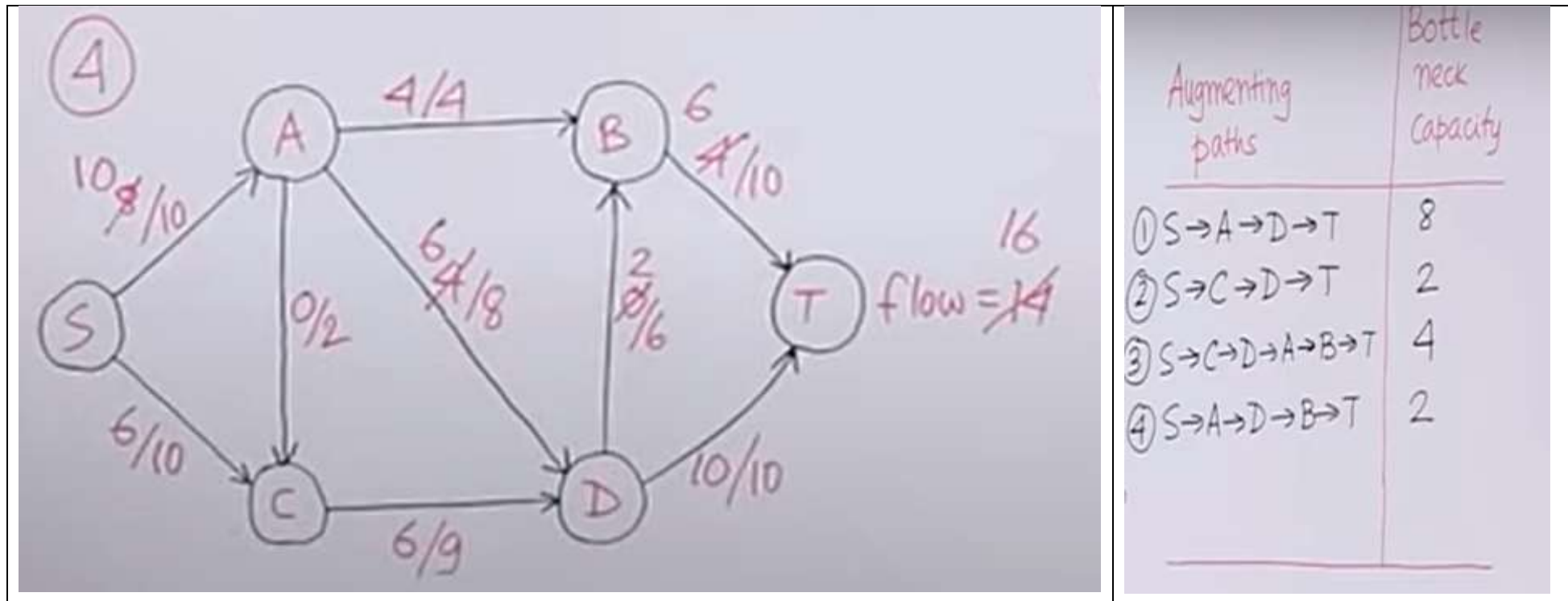
Updated figure is given here





Flow is  $10 + 4 = 14$

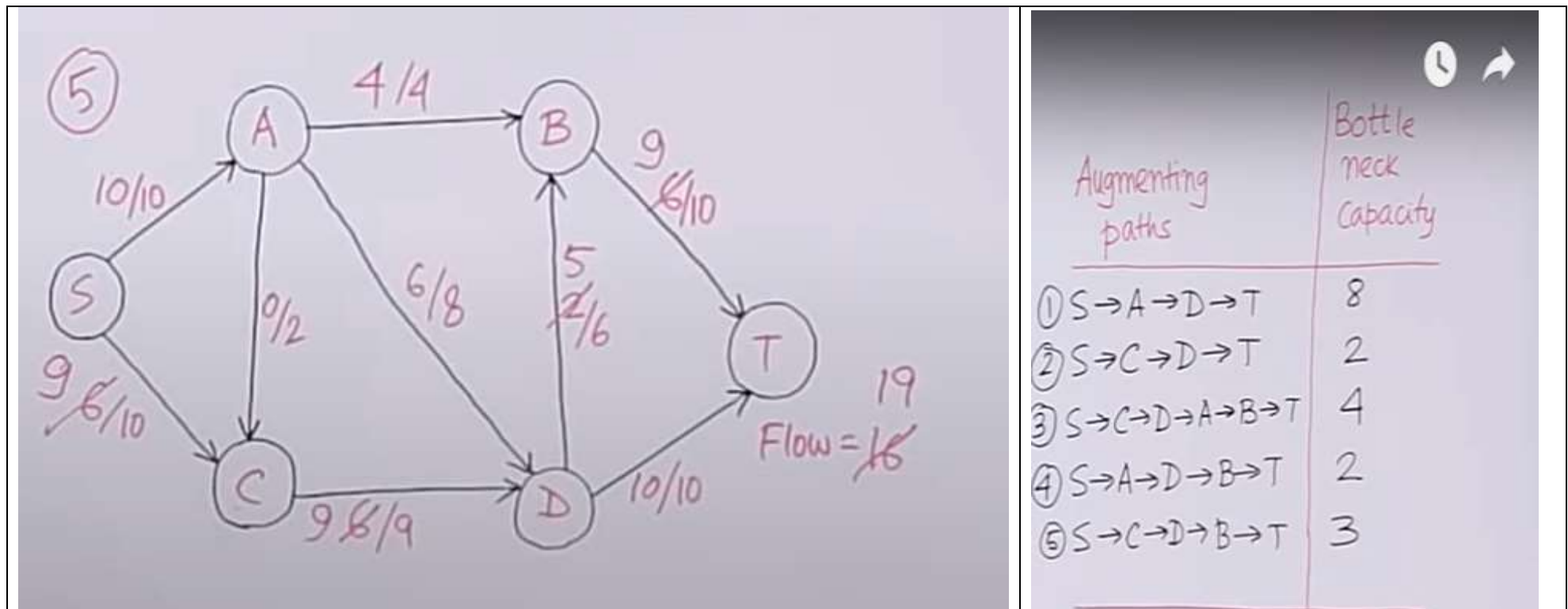
Next augmenting path is S → A → D → B → T with Bottle neck capacity 2 (2, 4, 6, 6)



Flow is  $14 + 2 = 16$

Next path selected is  $S \rightarrow C \rightarrow D \rightarrow B \rightarrow T$

Bottle neck capacity is 3 (4, 3, 4, 4)



Flow is  $16 + 3 = 19$

There is no augmenting path in the above figure. Flow of all the incoming flow is equal to the outgoing flow.

The basic Ford-Fulkerson algorithm

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

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

In each iteration of the Ford-Fulkerson method, we find some augmenting path  $p$  and use  $p$  to modify the flow  $f$ .

replace  $\tilde{f}$  by  $f \uparrow \tilde{f}_p$ , obtaining a new flow whose value is  $|f| + |\tilde{f}_p|$ . The following implementation of the method computes the maximum flow in a flow network  $G = (V, E)$  by updating the flow attribute  $(u, v).f$  for each edge  $(u, v) \in E$ .<sup>1</sup> If  $(u, v) \notin E$ , we assume implicitly that  $(u, v).f = 0$ . We also assume that we are given the capacities  $c(u, v)$  along with the flow network, and  $c(u, v) = 0$  if  $(u, v) \notin E$ . We compute the residual capacity  $c_f(u, v)$  in accordance with the formula (26.2). The expression  $c_f(p)$  in the code is just a temporary variable that stores the residual capacity of the path  $p$ .

## Complexity of the Algorithm

### Ford-Fulkerson algorithm steps

1. Find an augmenting path
2. Compute the bottle neck capacity
3. Augment each edge and total flow

### Ford-Fulkerson algorithm steps

Find an augmenting path

Compute the bottle neck capacity

Augment each edge and total flow

$O(E)$

$E = \text{No. of edges.}$

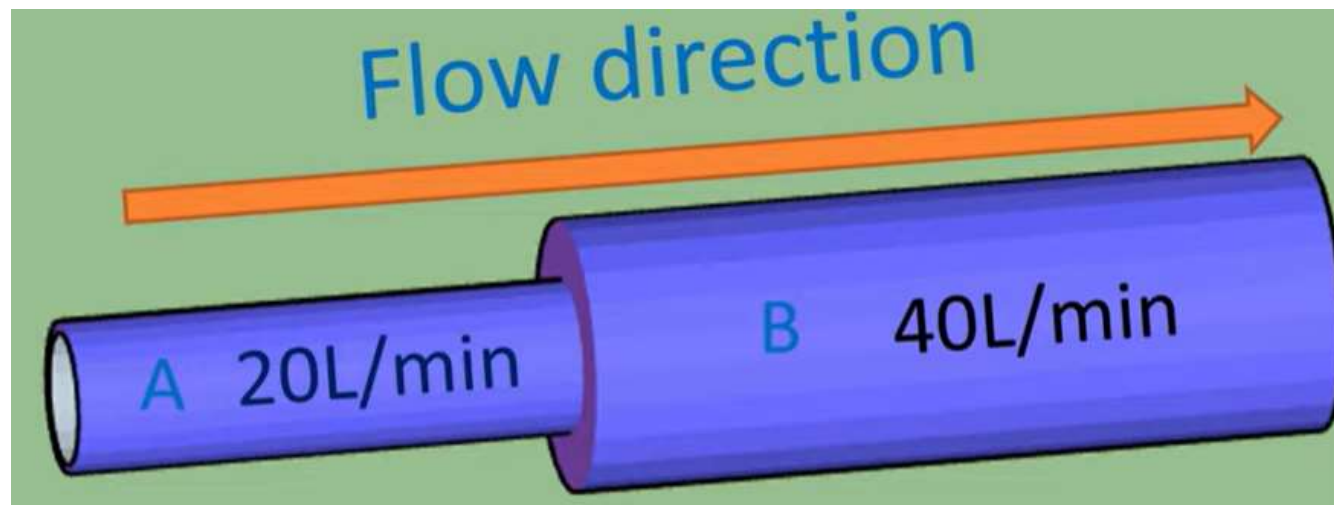
$F = \text{Maximum Flow.}$

$O(F)$

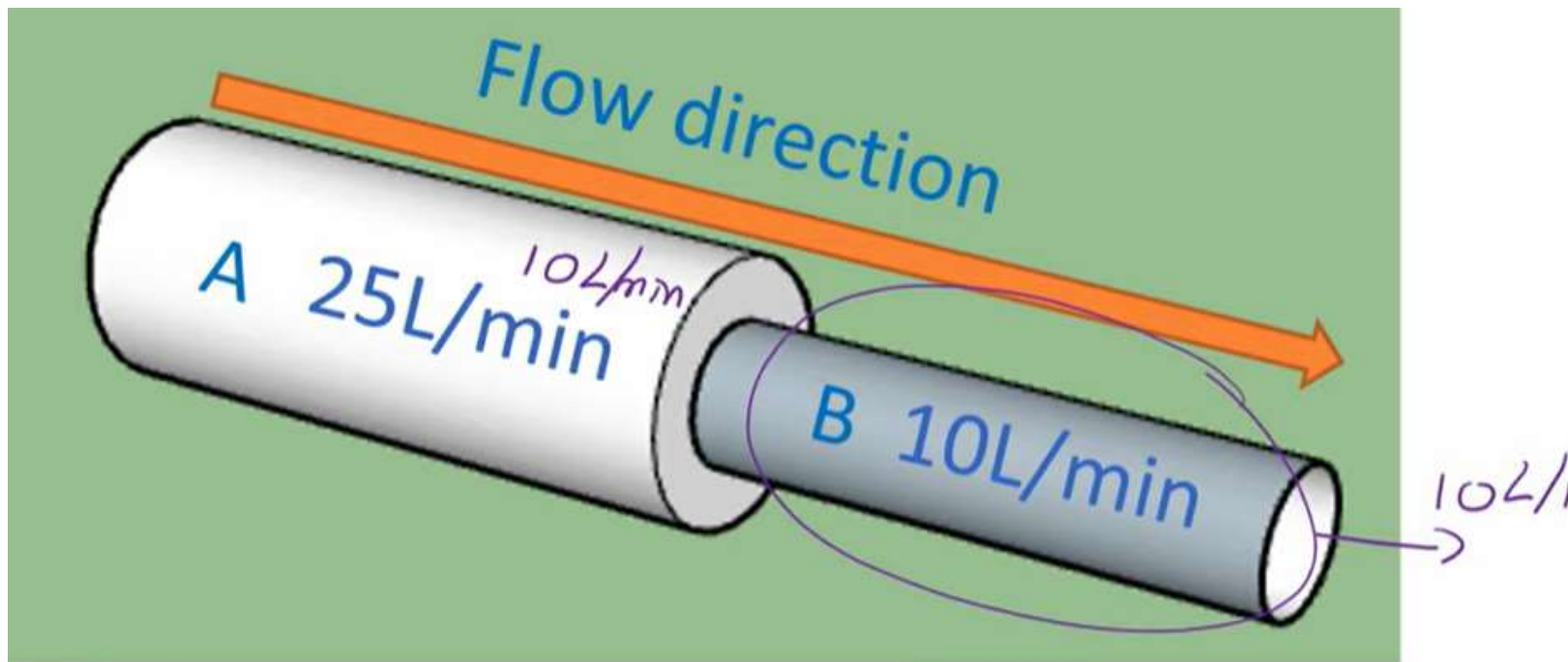
$O(E * F)$

## Max- Flow Min Cut Theorem

In the maximum-flow problem, we wish to compute the greatest rate at which we can ship material from the source to the sink without violating any capacity constraints. It is one of the simplest problems concerning flow networks.







The maximum-flow minimum-cut theorem

**Minimum cut = Maximum flow**

## Definition of a Cut

A cut (also called as st-cut) is the division of graph into two partitions A and B such that some nodes are in partition A and some nodes in partition

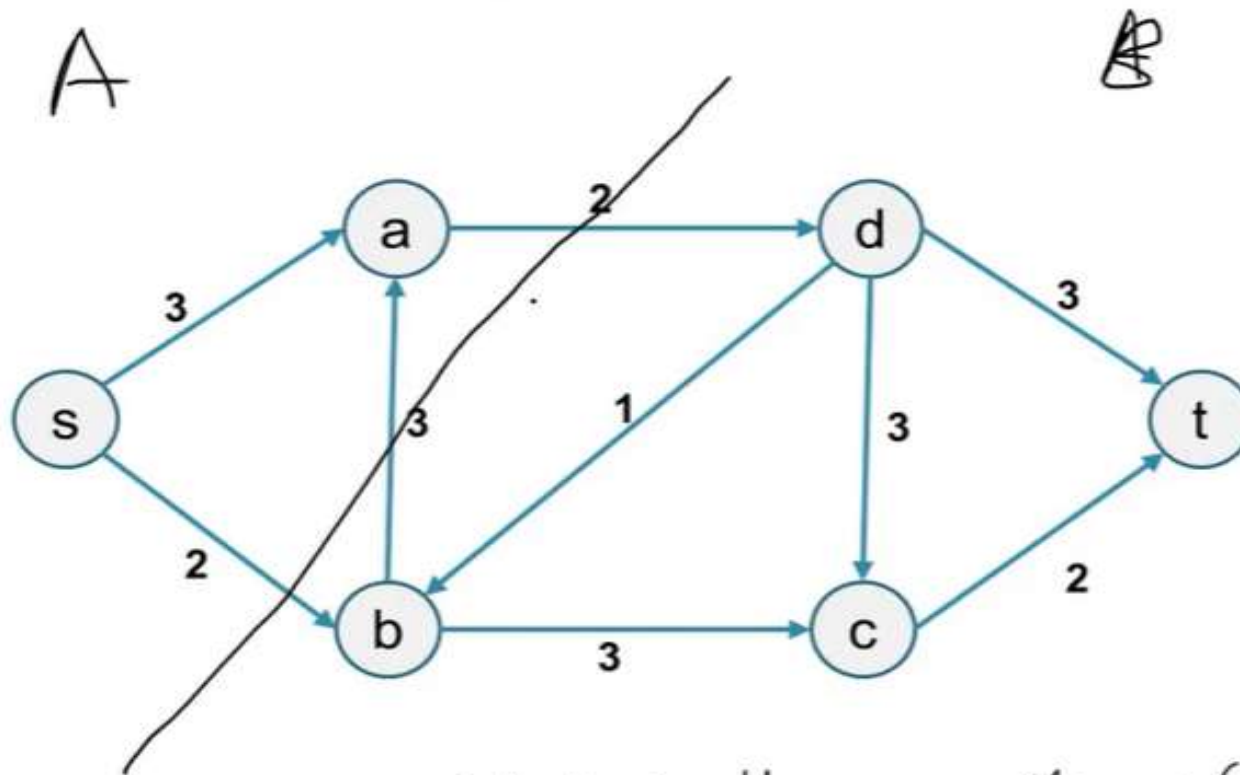


B and should satisfy the constraint that  $s$  must belongs to A while  $t$  must belongs to B.

The capacity of an  $s$ - $t$  cut is defined by the sum of the capacity of each edge in the cut-set.

The max-flow min-cut theorem states that in a flow network, the amount of maximum flow is equal to capacity of the minimum cut.

## Example of Cut - 1



$$A = \{s, a\}$$

$$B = \{b, c, d, t\}$$

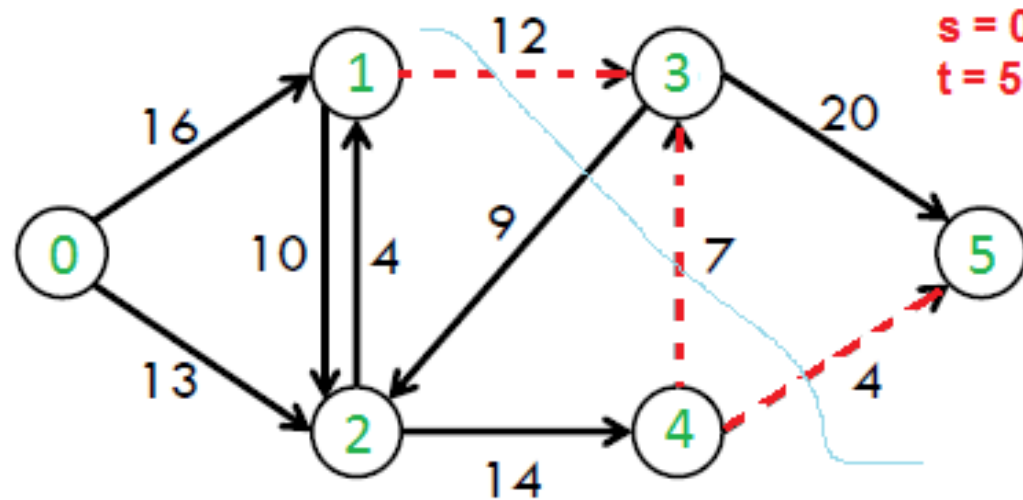
What is the capacity  $c(A, B) = \sum_{e \text{ out of } A} c(e)$

$$2 + 2 = 4$$

Summation of all the capacities of the edges going out of the cut. ( s to b and a to d)

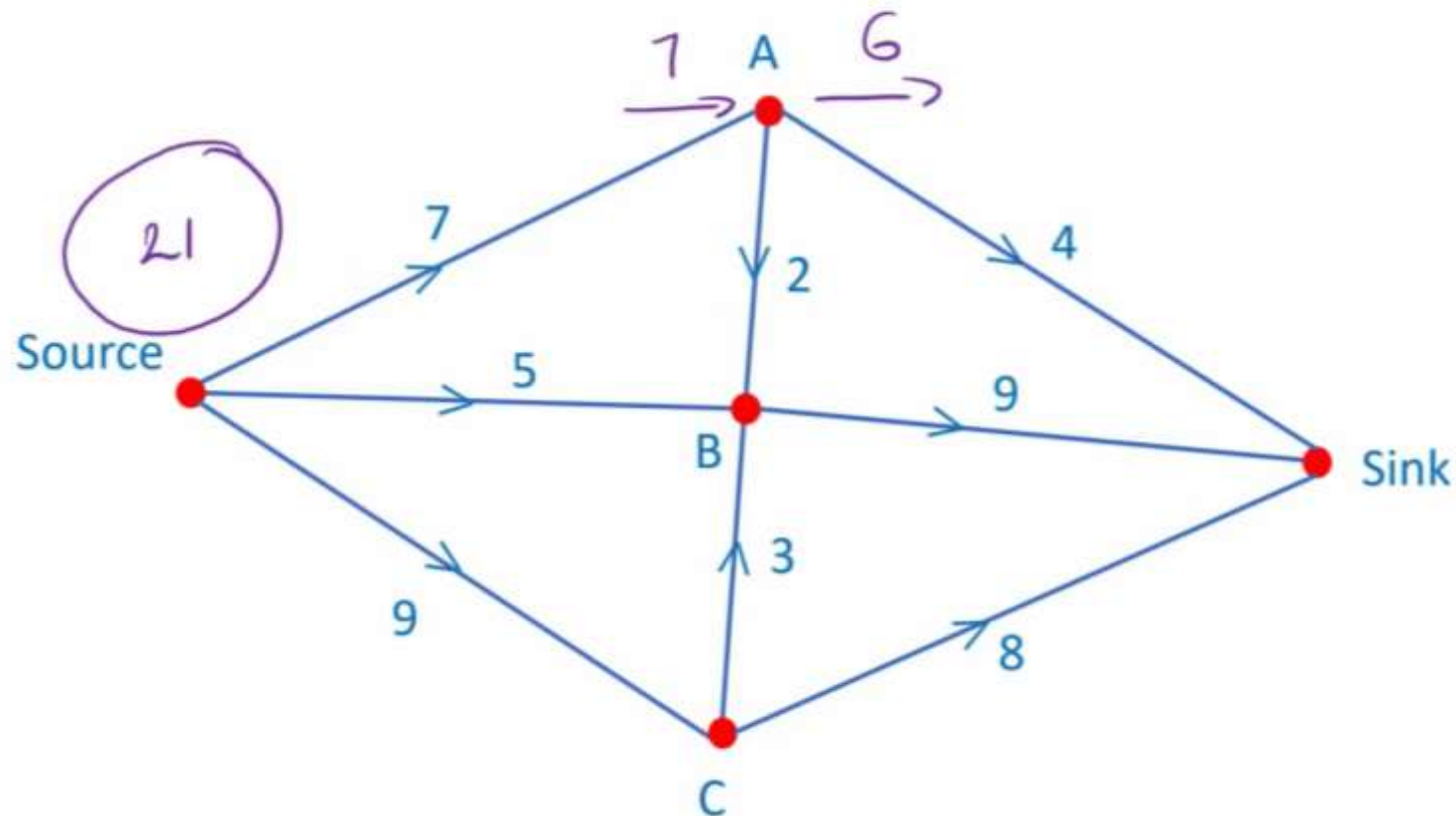
## Another example

In the following flow network, example s-t cuts are  $\{\{0, 1\}, \{0, 2\}\}$ ,  $\{\{0, 2\}, \{1, 2\}, \{1, 3\}\}$ , etc. The minimum s-t cut is  $\{\{1, 3\}, \{4, 3\}, \{4, 5\}\}$  which has capacity as  $12+7+4 = 23$ .



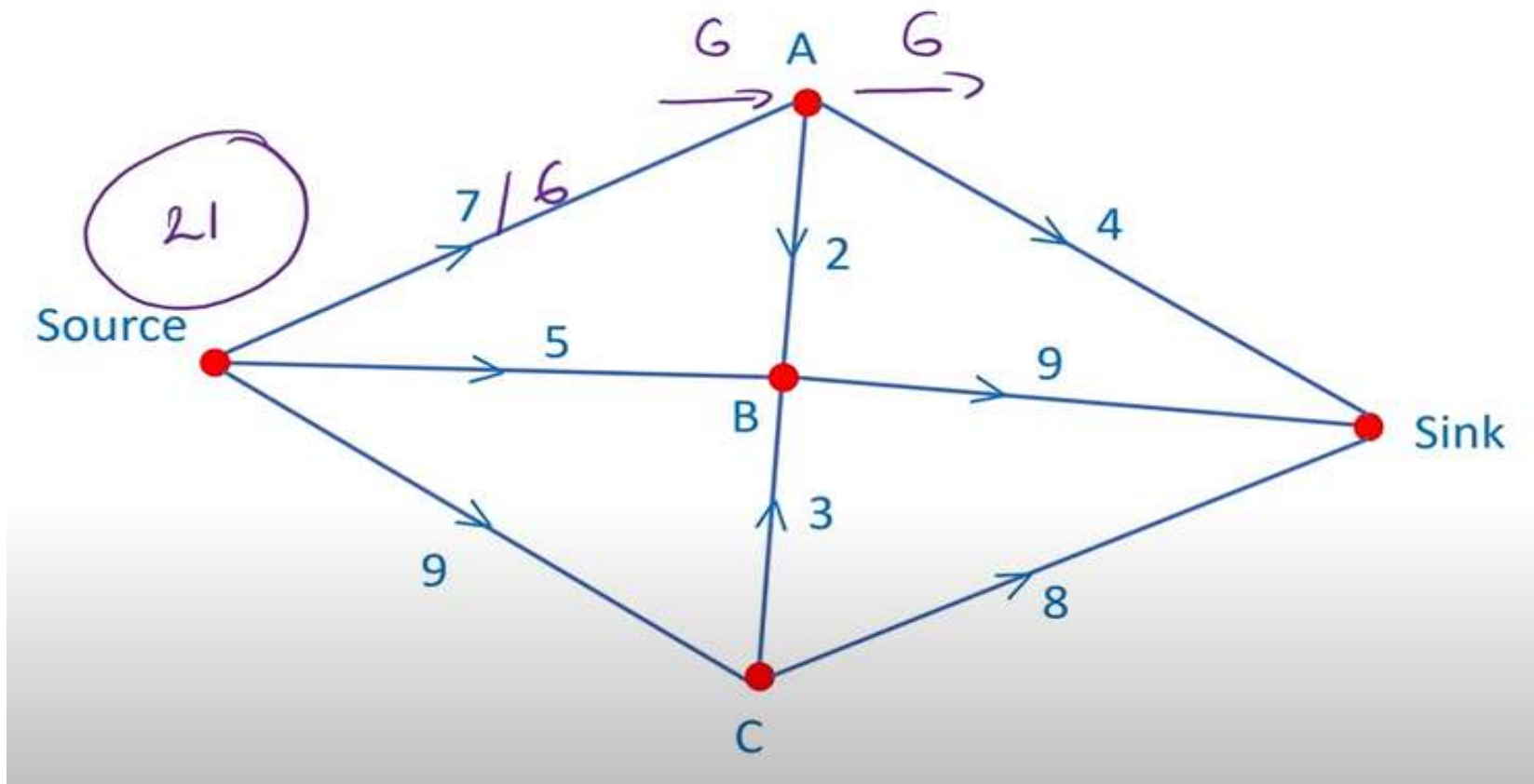
These ideas are essential to the important max-flow min-cut theorem

Find the maximum flow for the network diagram below.

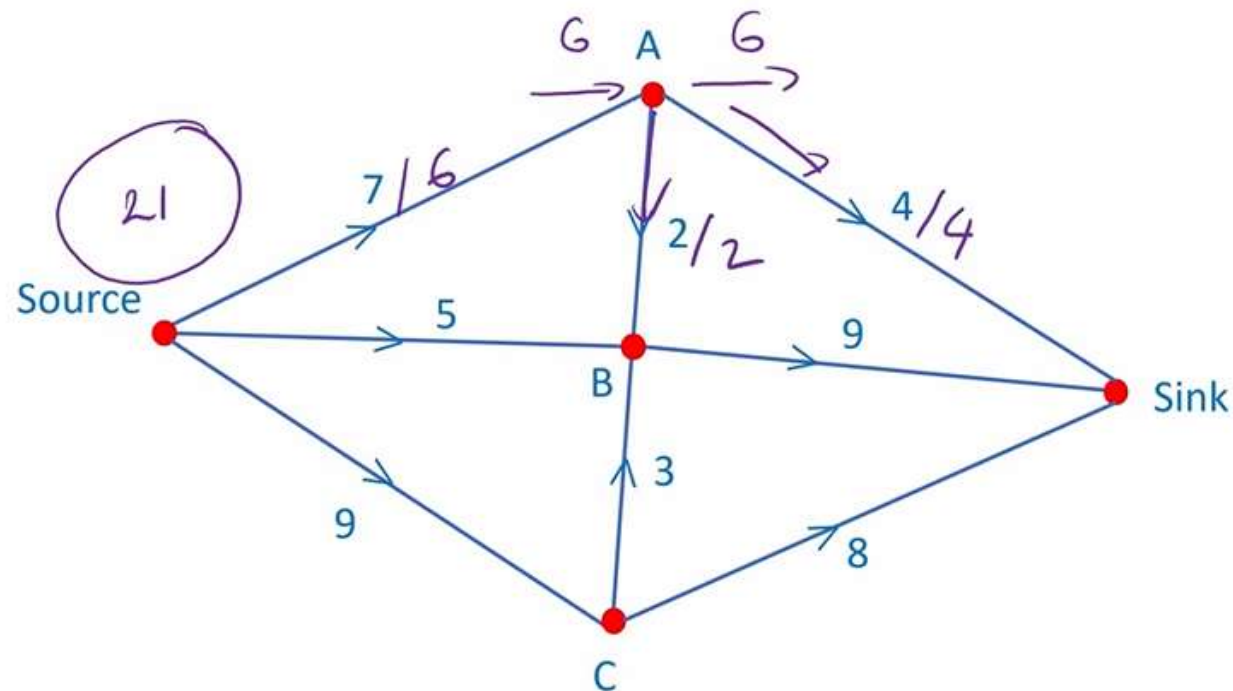


Consider the flow of node A, 7 is not possible to flow to 6 so changed the value

Find the maximum flow for the network diagram below.

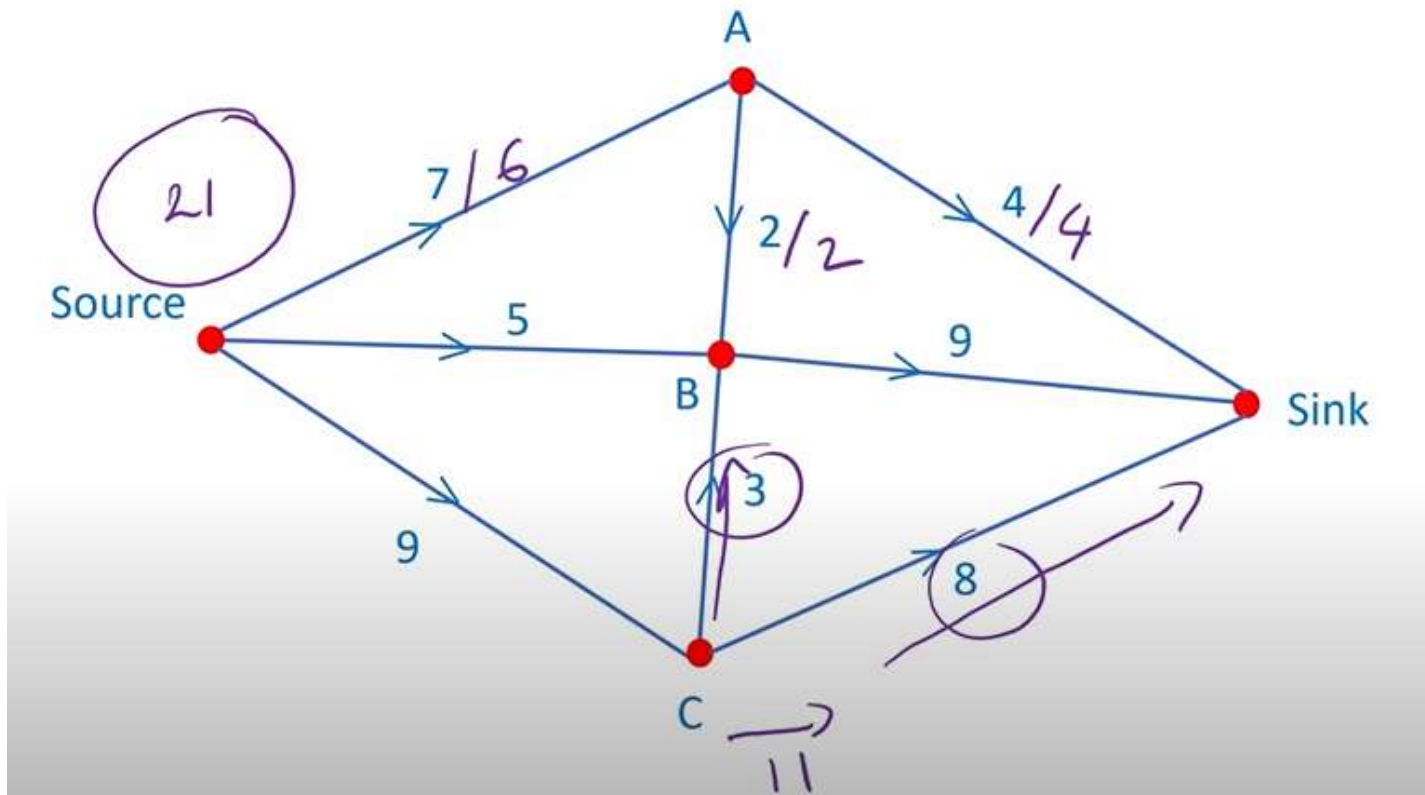


Find the maximum flow for the network diagram below.



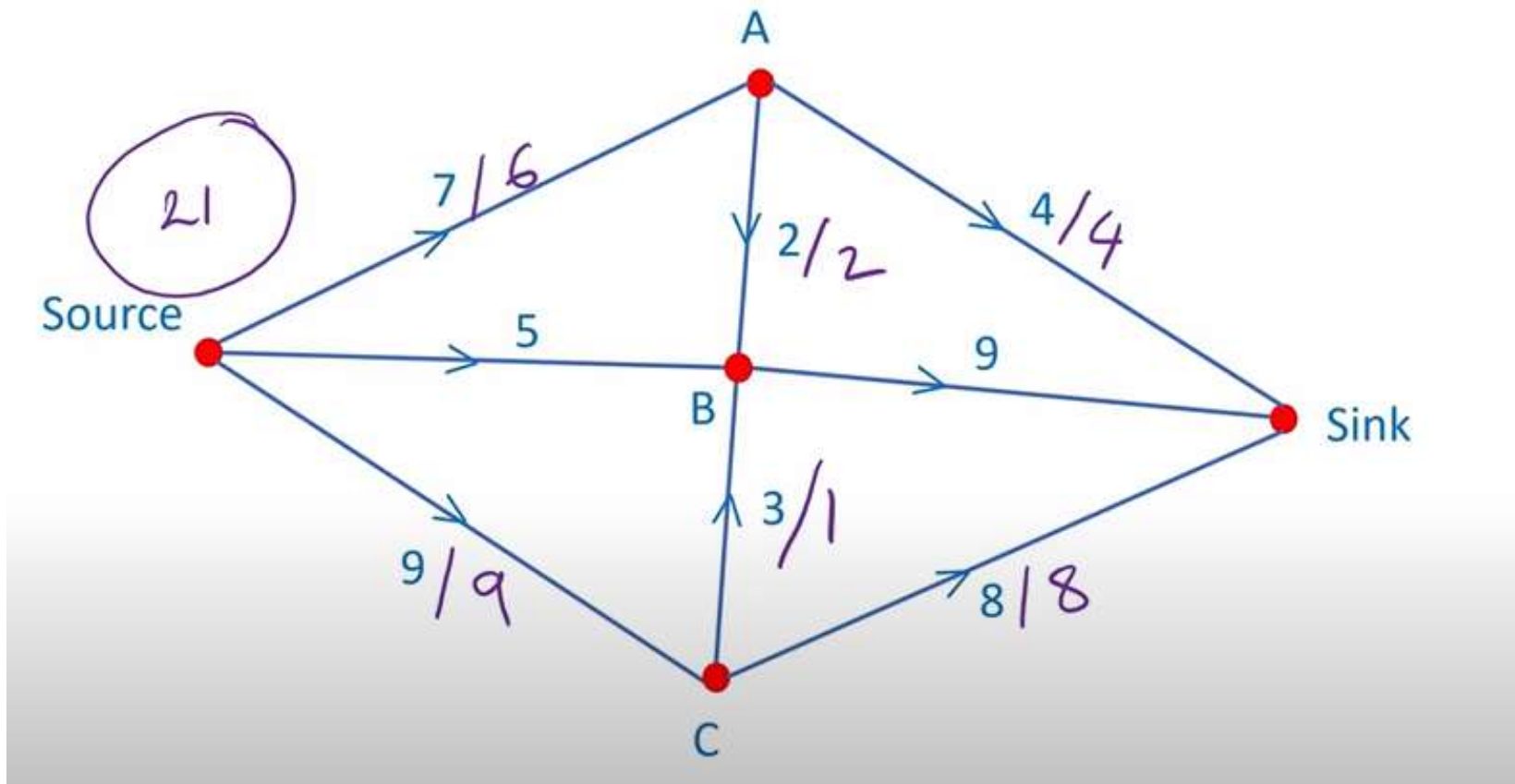
Flow from source to A is capacity 6 which can flow through A->sink (4) and A->B (2)

Find the maximum flow for the network diagram below.



The above figure 9 units flow into C, so maximum flow possible is only 9.

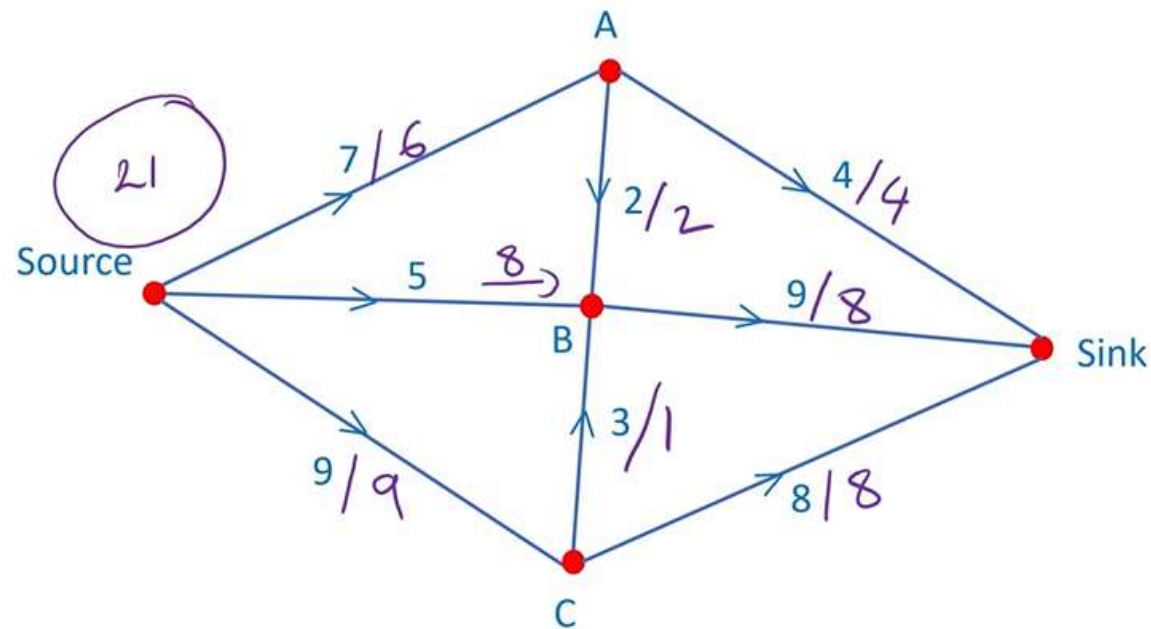
Find the maximum flow for the network diagram below.



In the following figure, in vertex B, possible maximum inflow and outflow is 8.

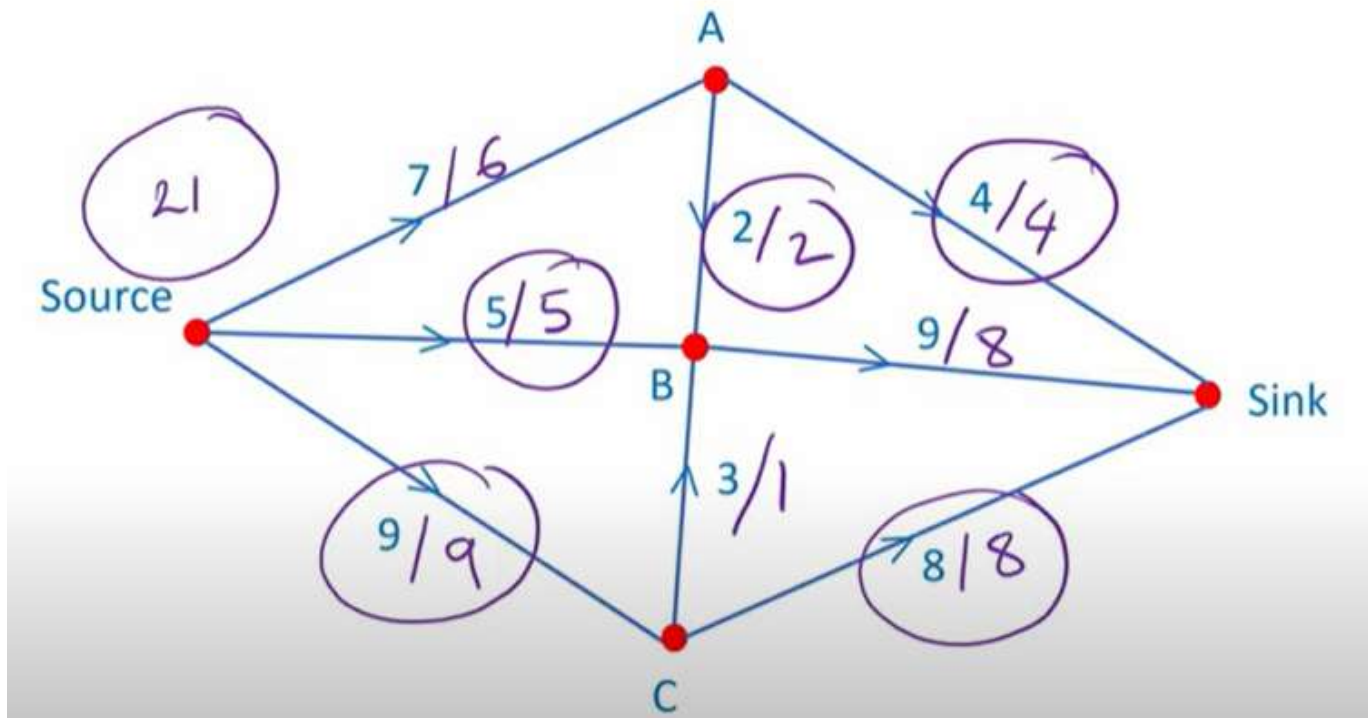


Find the maximum flow for the network diagram below.



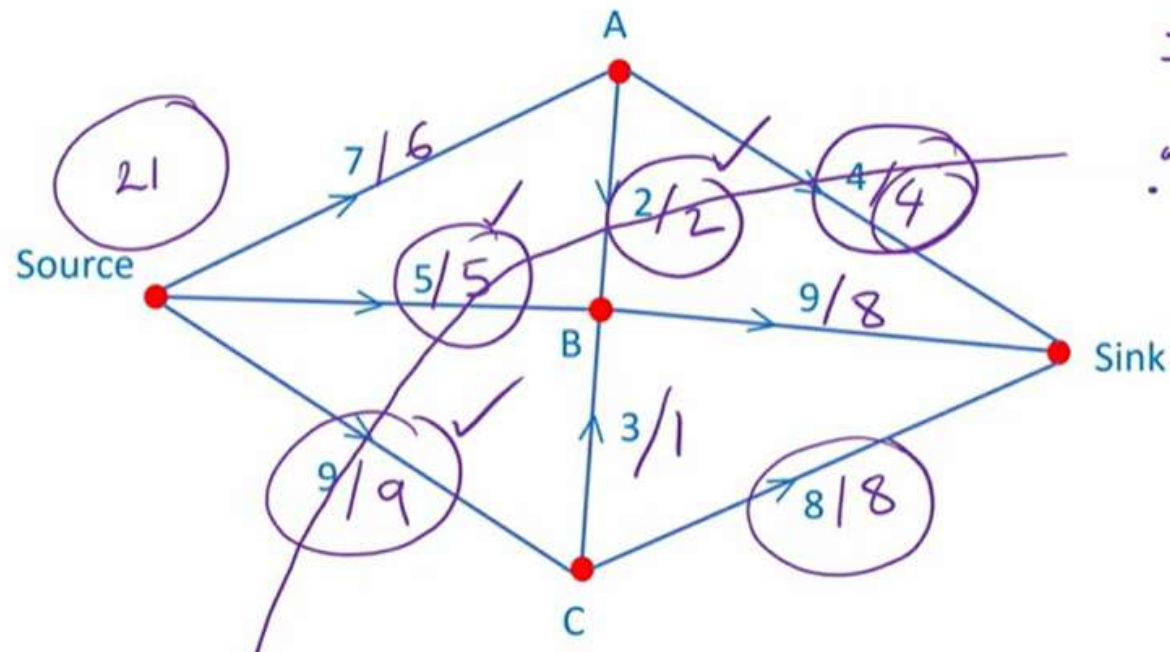
In the given graph maximum flow nodes are

Find the maximum flow for the network diagram below.



Cut the graph through the maximum flow edges which can shown in the figure

Find the maximum flow for the network diagram below.

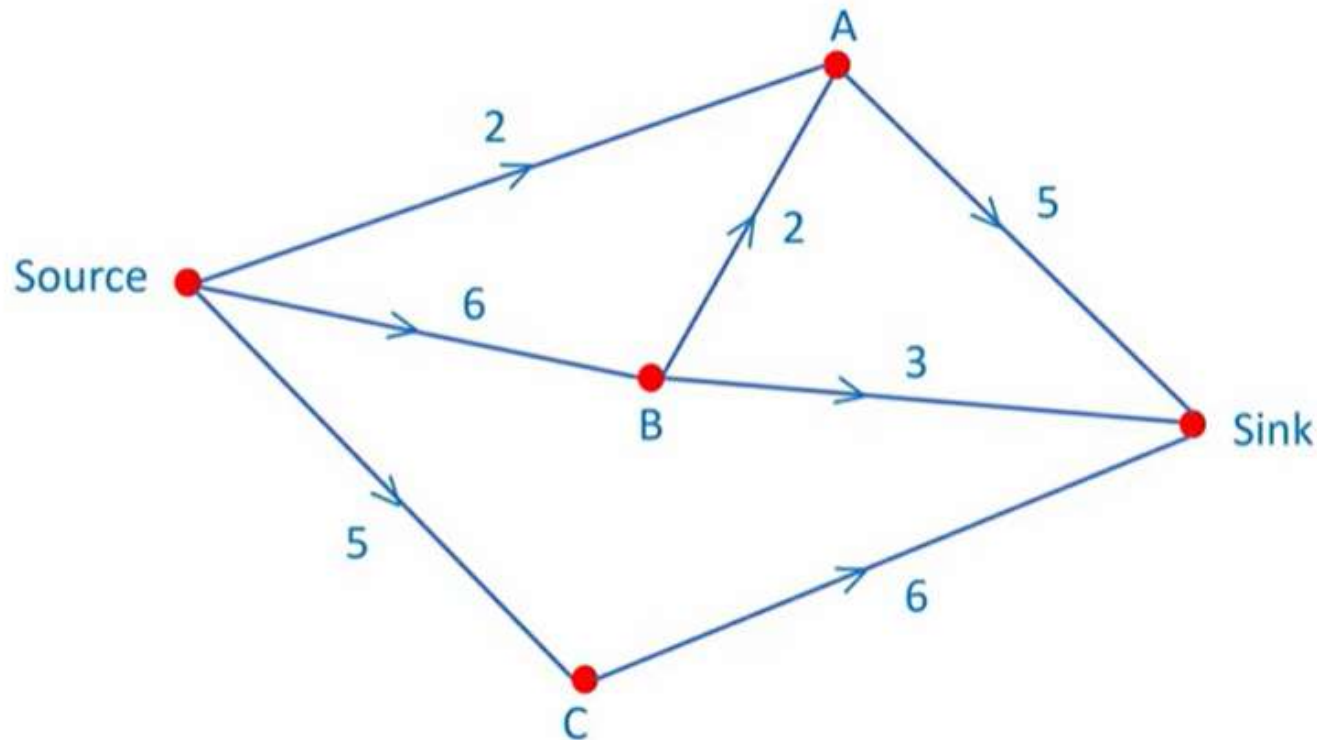


Ignore the edge C  $\rightarrow$  Sink.

Maximum capacity is 20.

**Example 2 (home work)**

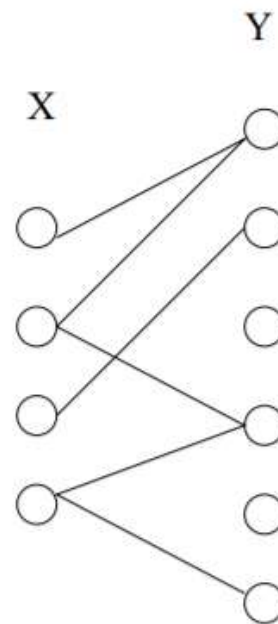
Find the maximum flow for the network diagram below.



## Bipartite Matching

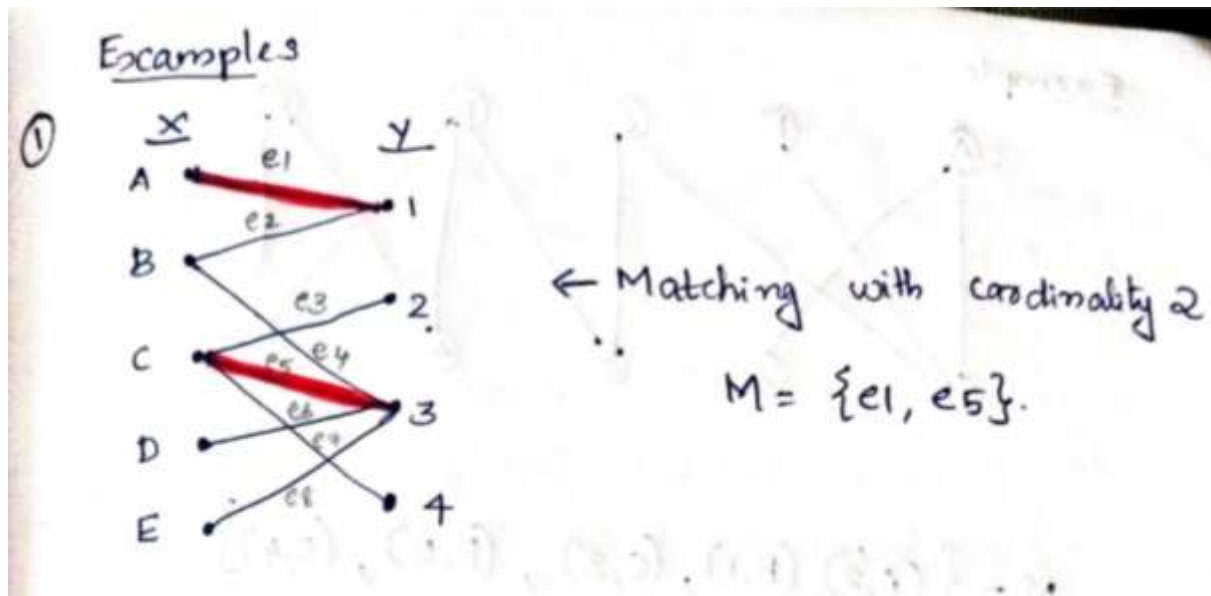
A Bipartite Graph  $G = (V, E)$  is a graph in which the vertex set  $V$  can be divided into two disjoint subsets  $X$  and  $Y$  such that every edge  $e \in E$  has

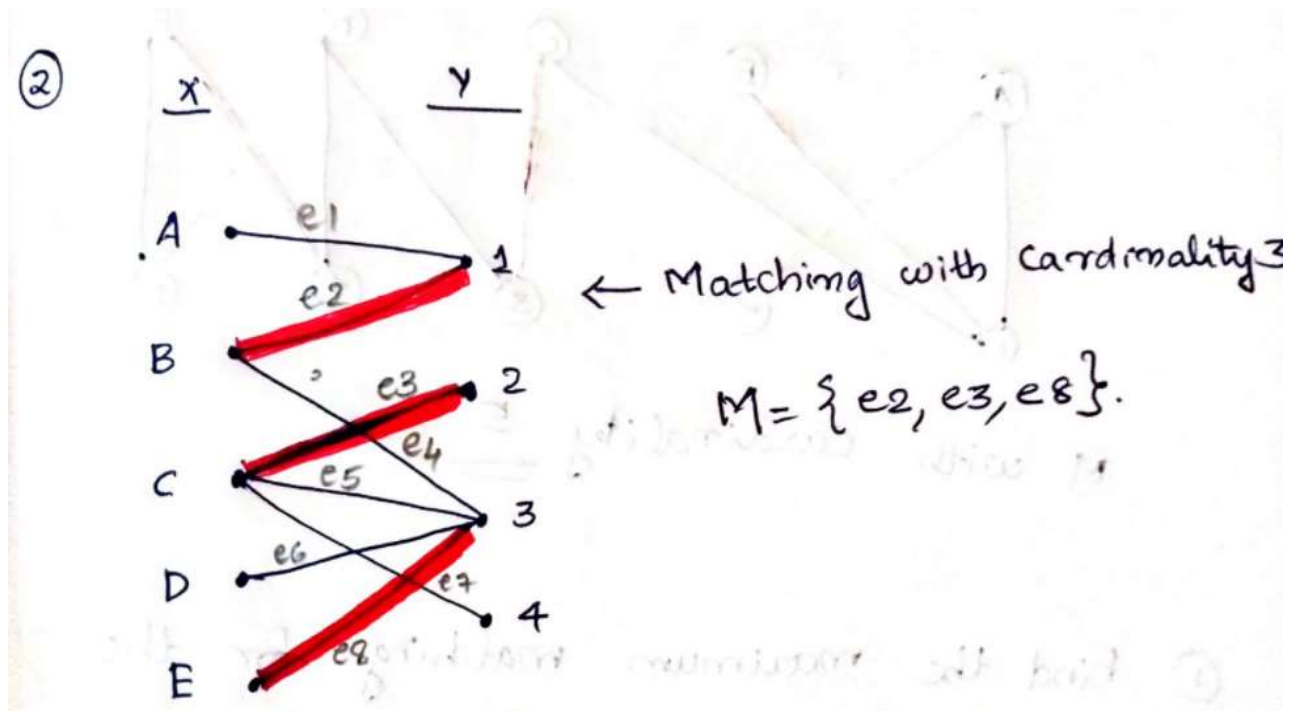
one end point in  $X$  and the other end point in  $Y$ . A matching  $M$  is a subset of edges such that each node in  $V$  appears in at most one edge in  $M$ . (no two edges share a vertex)



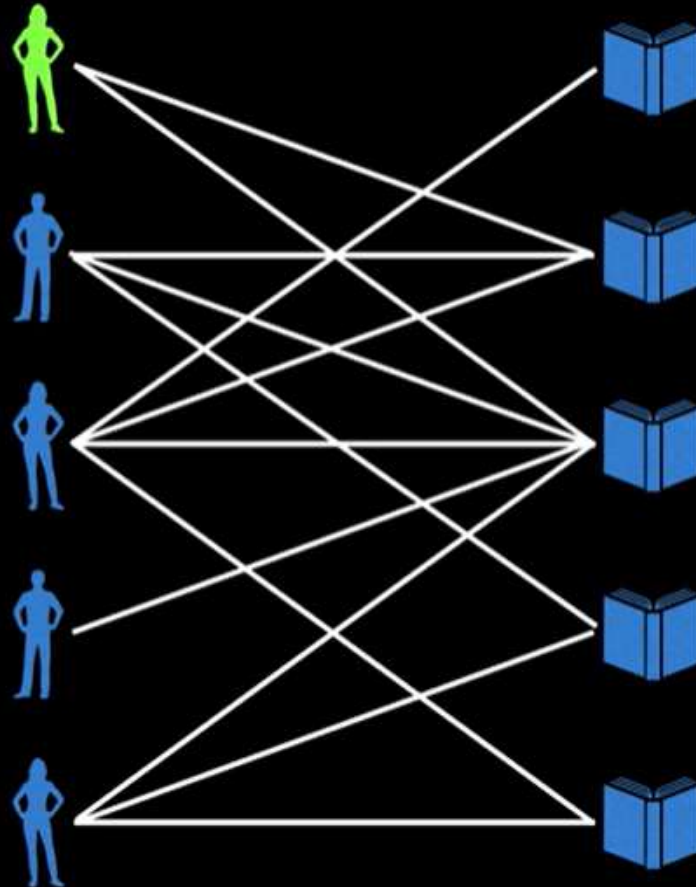
We are interested in matching of large size. Formally, maximal and maximum matching are defined as follows. (Maximal Matching) A

maximal matching is a matching to which no more edges can be added without increasing the degree of one of the nodes to two; it is a local maximum. (Maximum Matching) A maximum matching is a matching with the largest possible number of edges; it is globally optimal. Our goal is to find the maximum matching in a graph. Note that a maximal matching can be found very easily — just keep adding edges to the matching until no more can be added. Moreover, it can be shown that for any maximal matching  $M$ , we have that  $|M| \geq \frac{1}{2} |M^*|$  where  $M^*$  is the maximum matching.



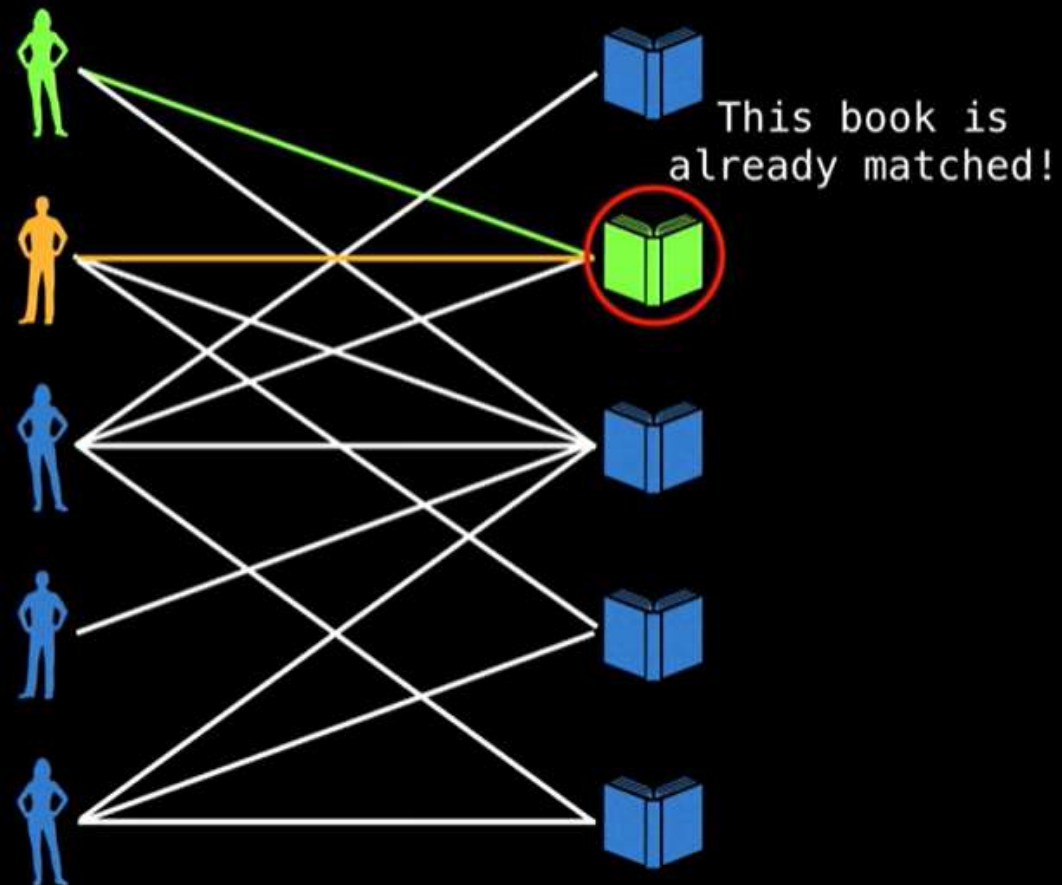


Let's try a greedy matching solution

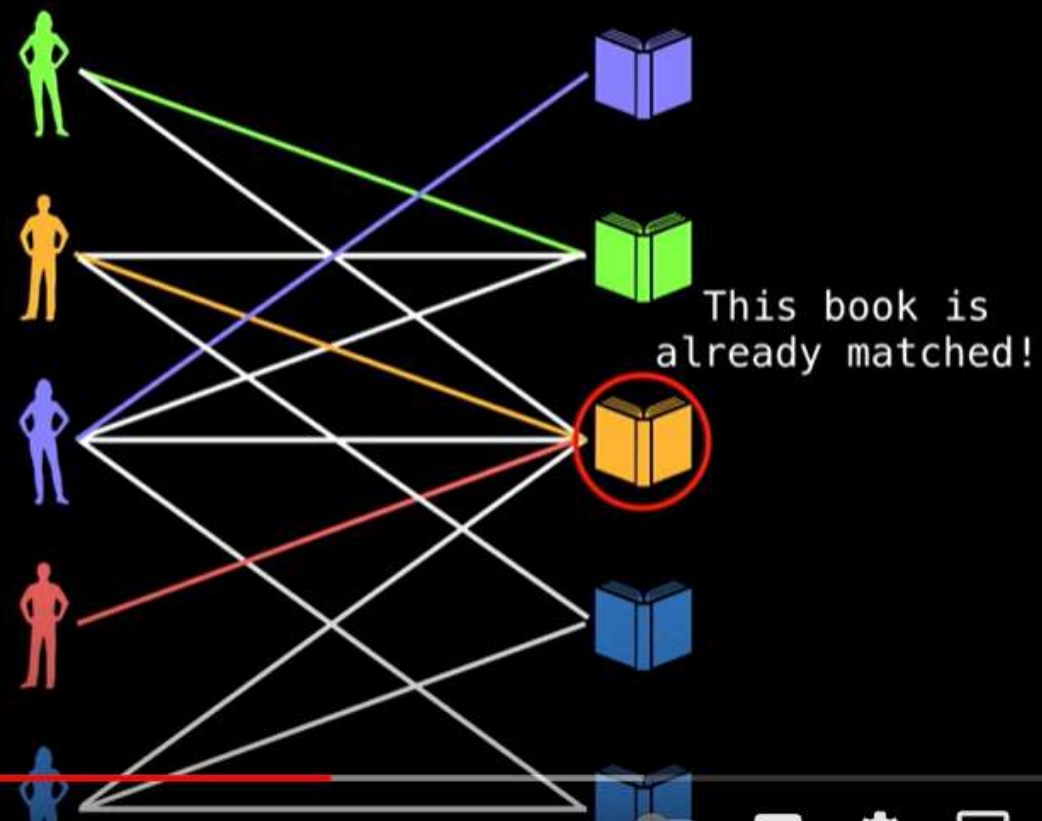




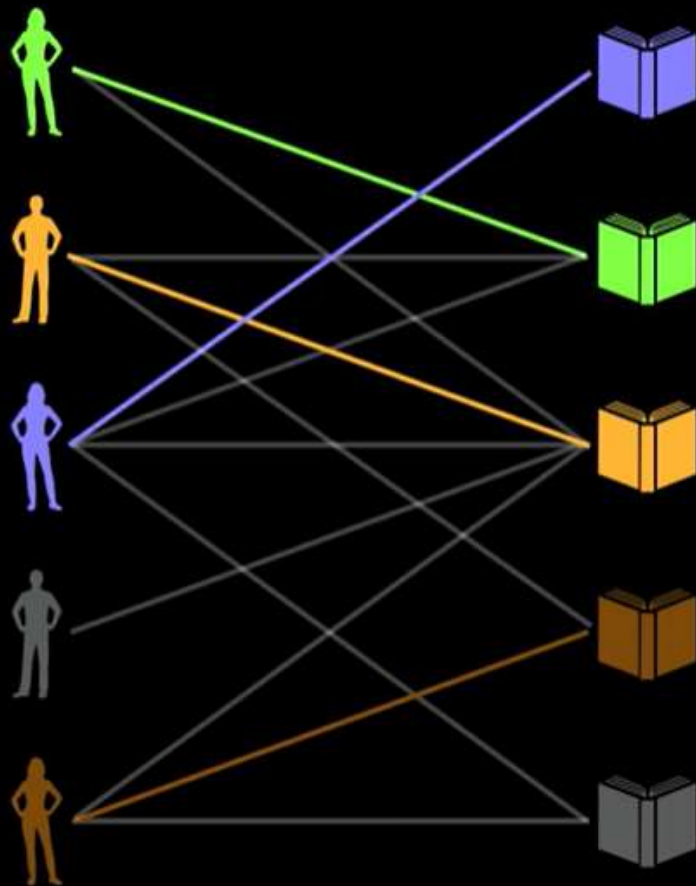
Let's try a greedy matching solution



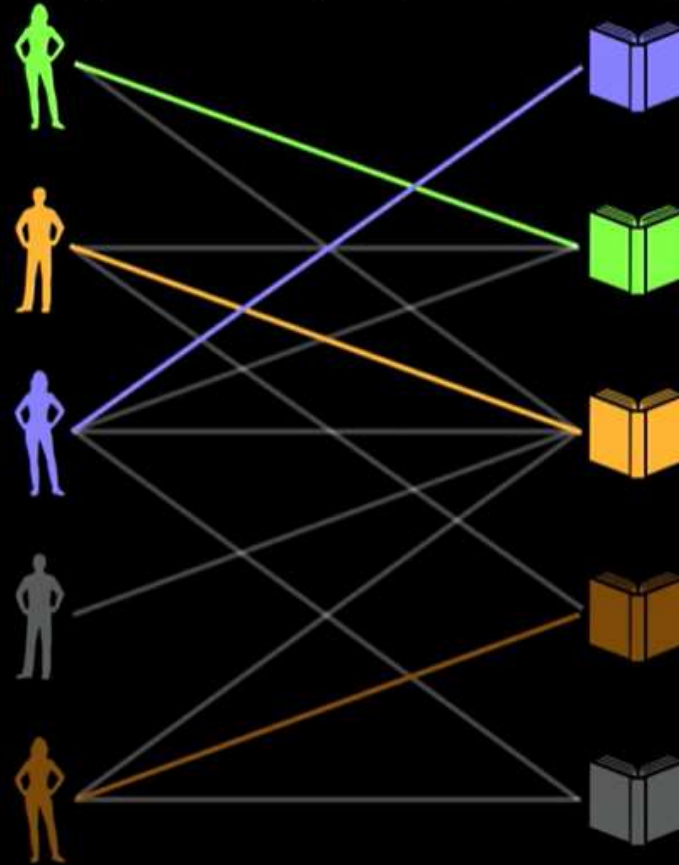
Let's try a greedy matching solution



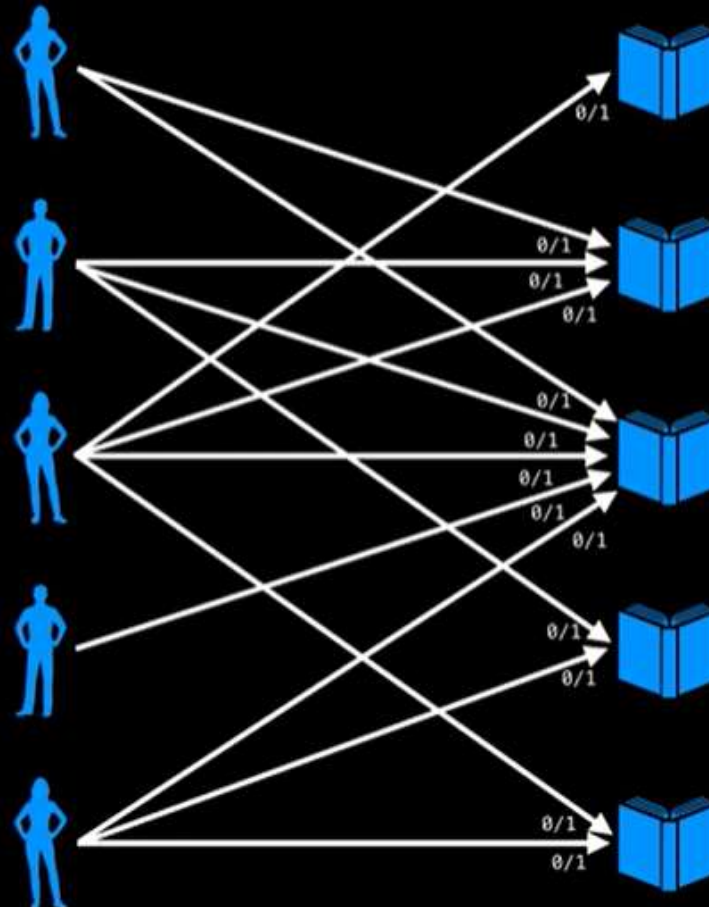
The greedy approach found 4 matchings,  
but can we do better?



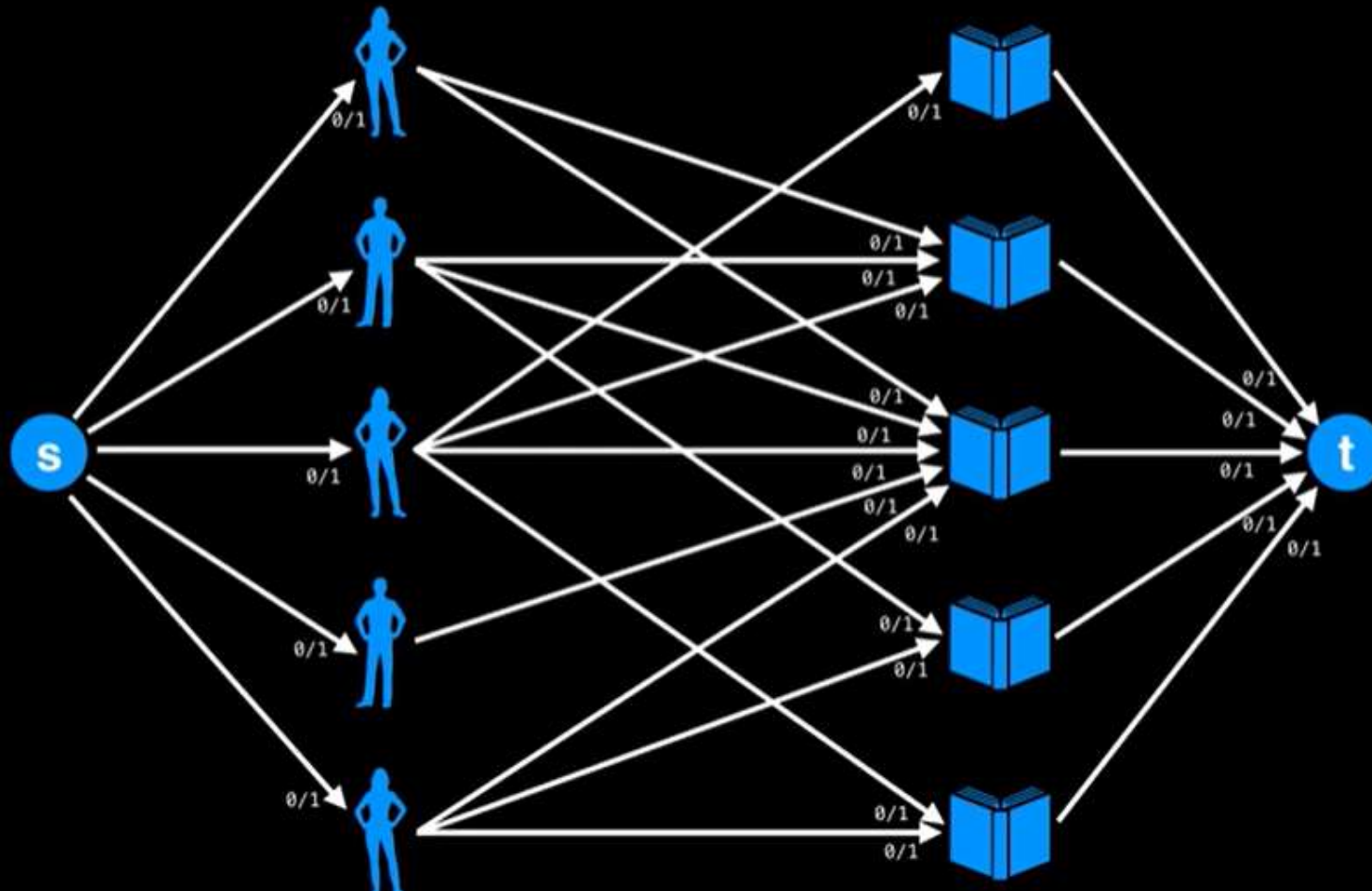
Yes! We can turn this matching problem into a network flow problem by adding a sink  $s$  and a source  $t$  and making each edge have unit capacities.



First make the edges directed and add unit capacities to each edge. The 0/1 besides each edge means 0 flow and a maximum capacity of 1.



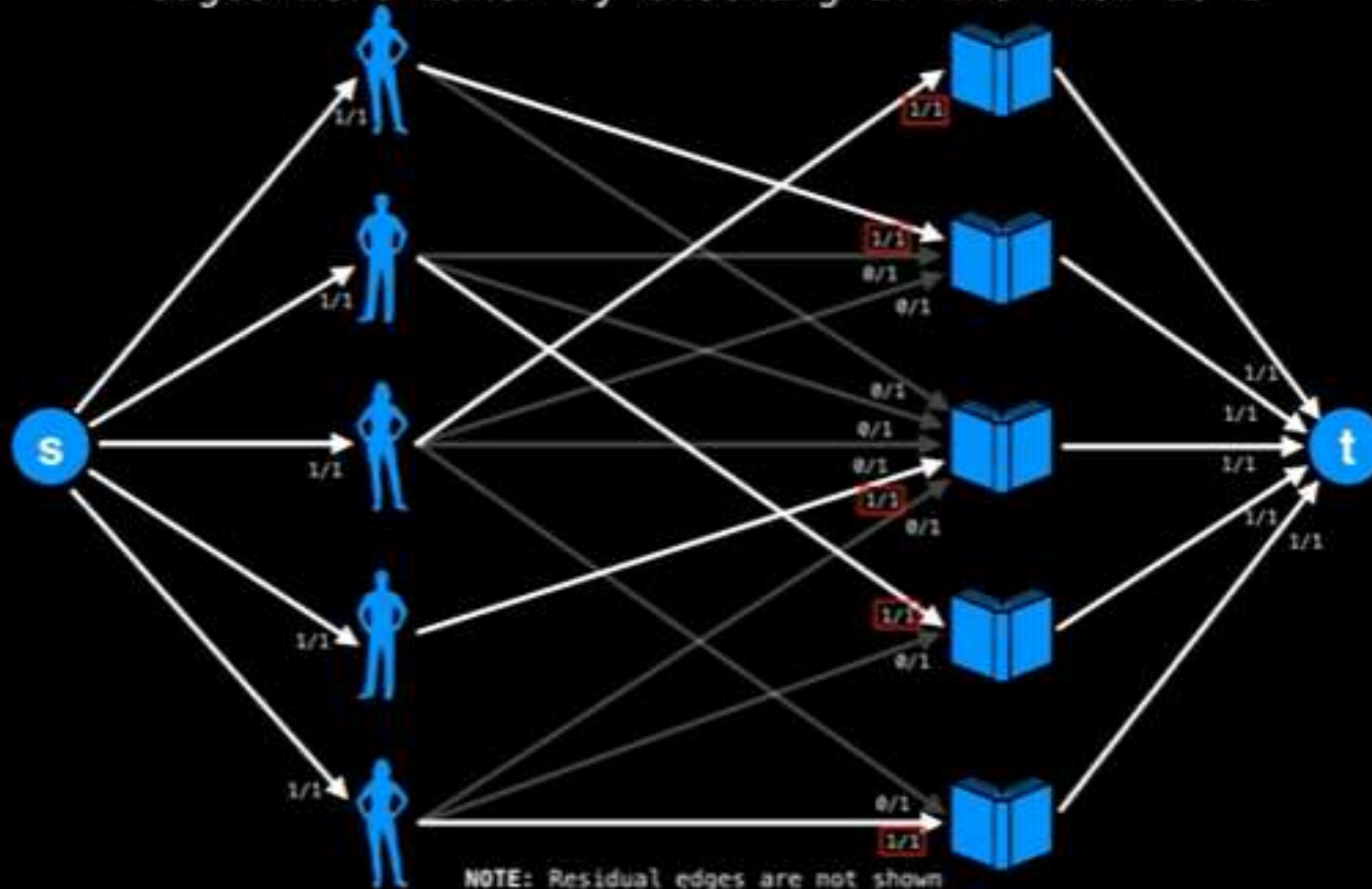
Once the flow graph is set up, use any max-flow algorithm to push flow through the network.



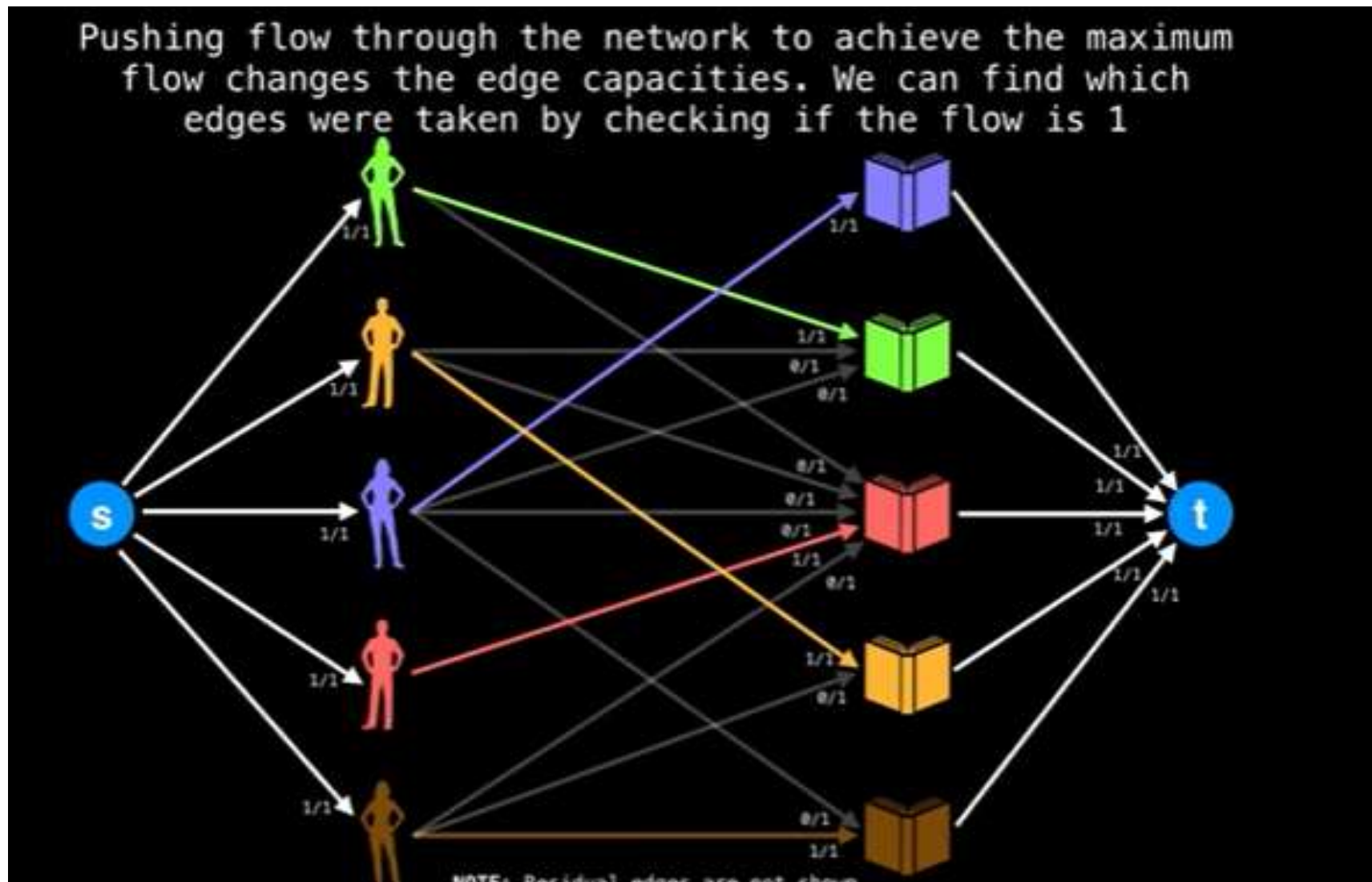




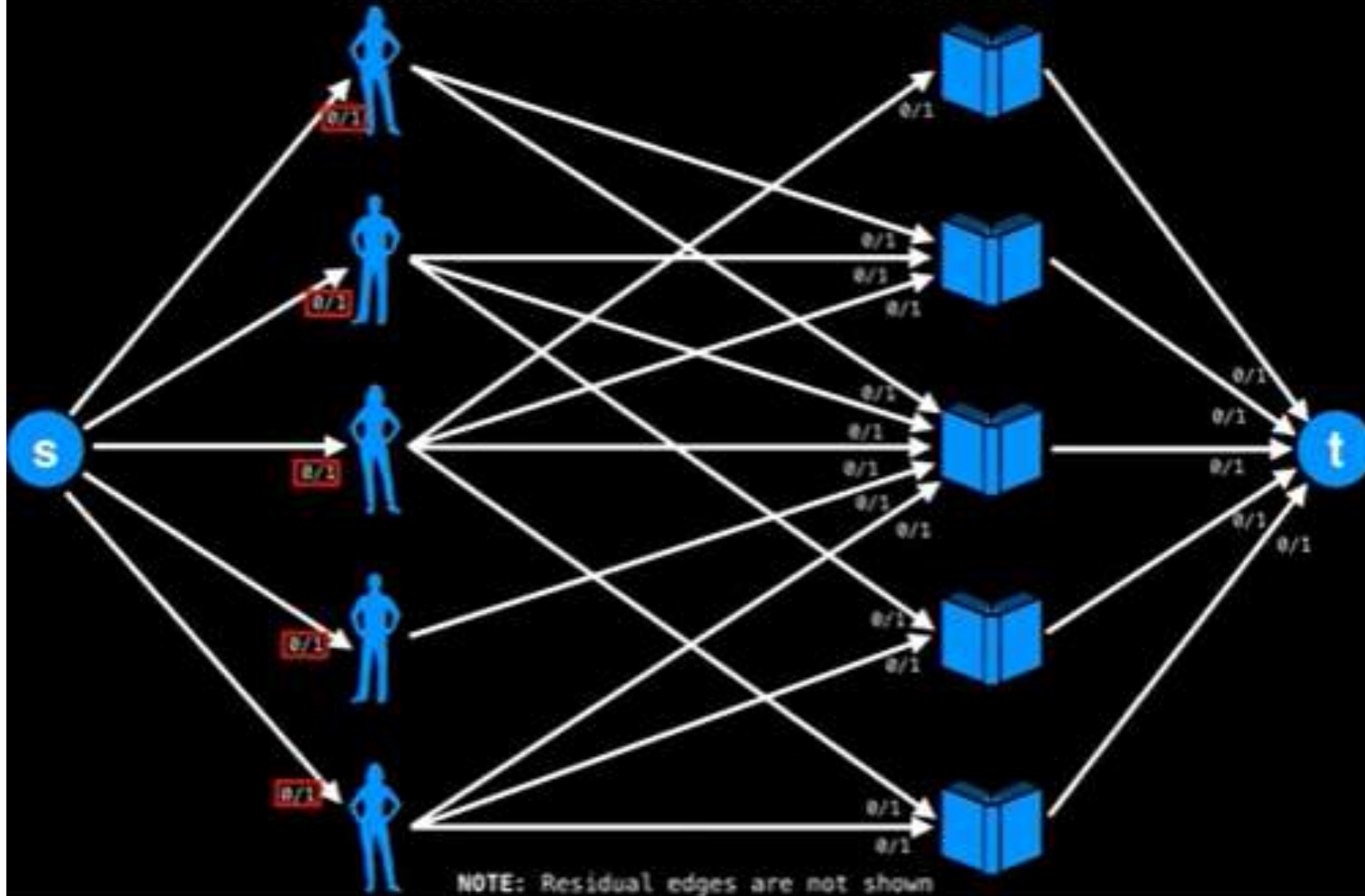
Pushing flow through the network to achieve the maximum flow changes the edge capacities. We can find which edges were taken by checking if the flow is 1

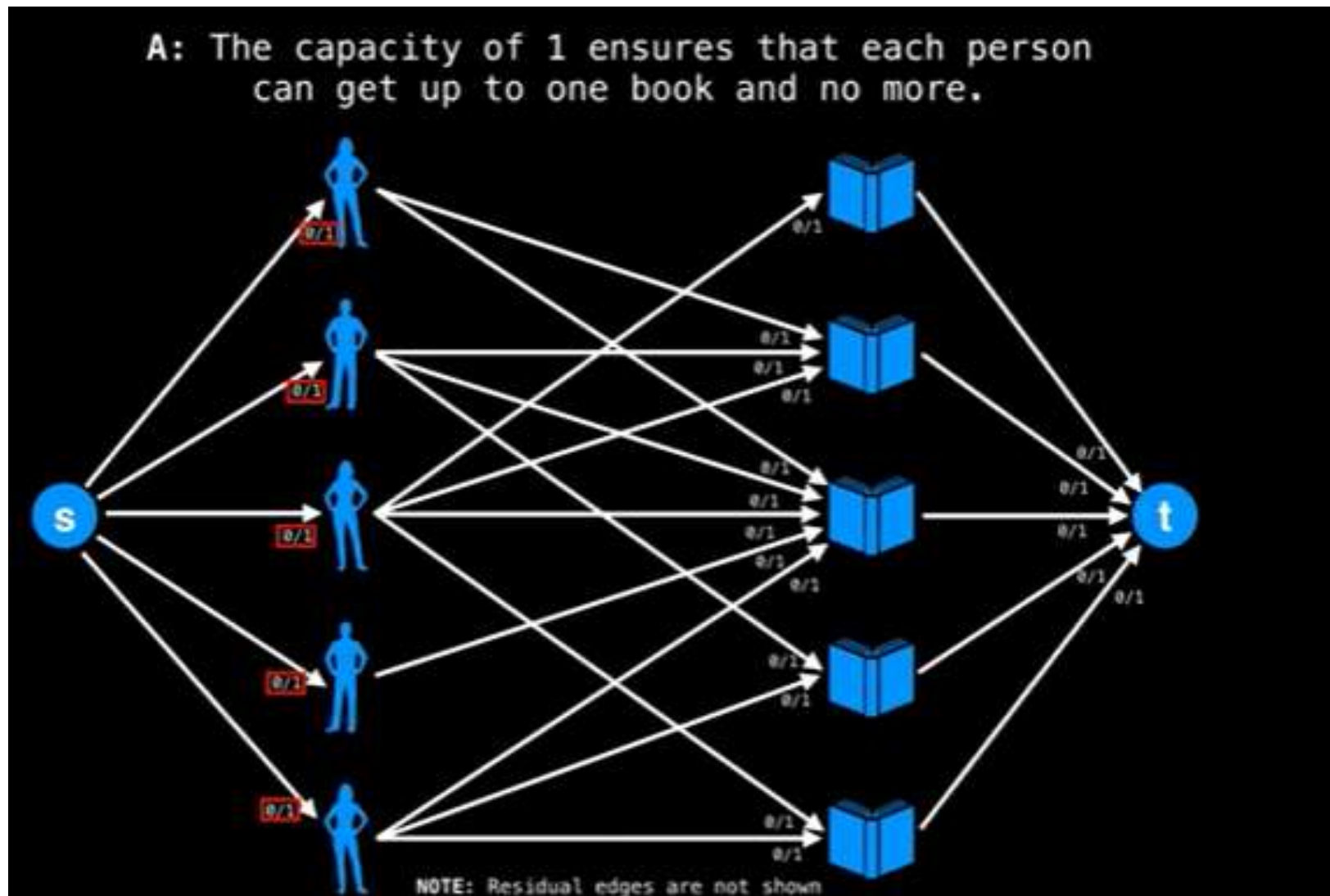


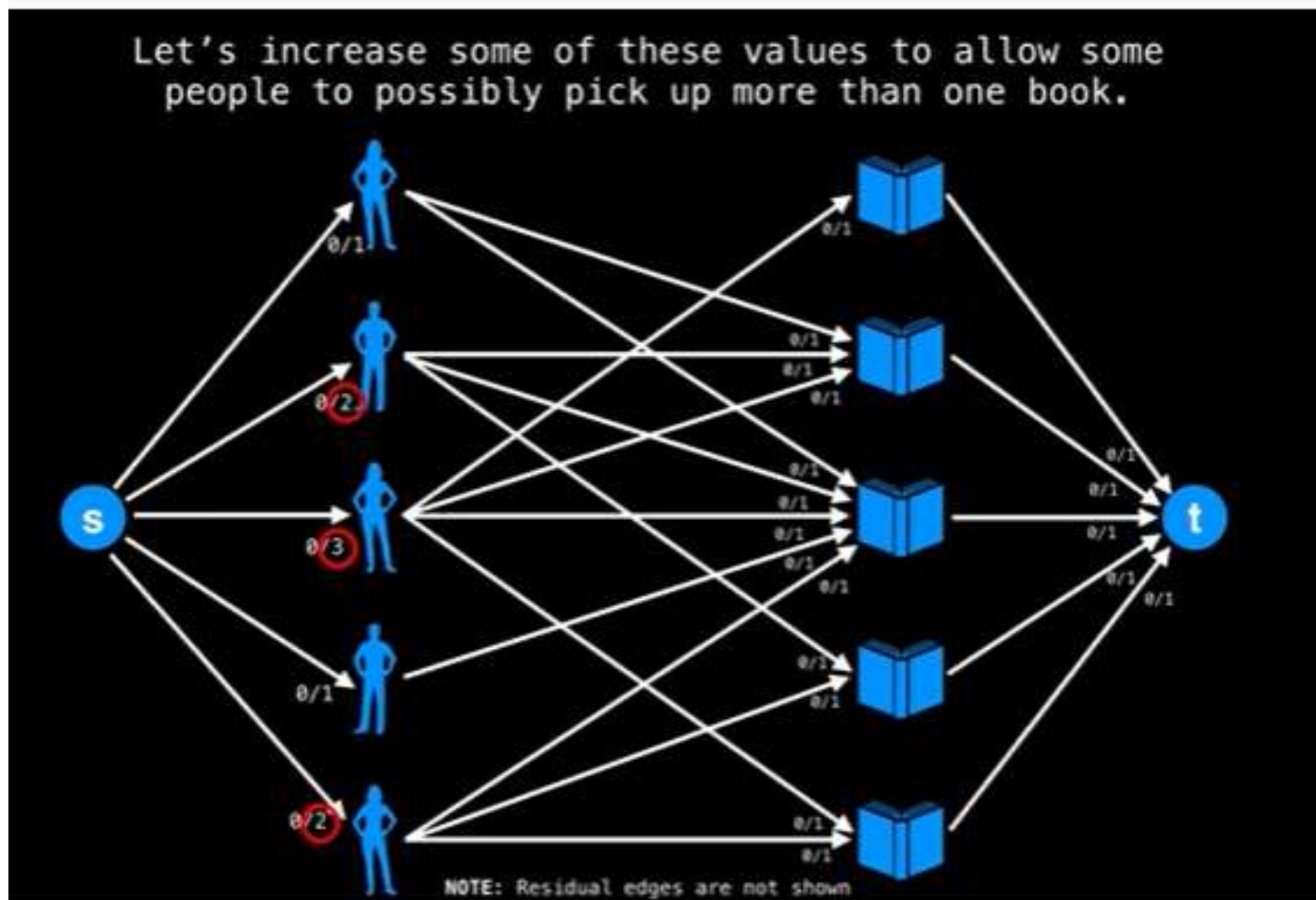




Q: We originally set the capacity of each edge from the source to each person to be 1, what constraint does this enforce?

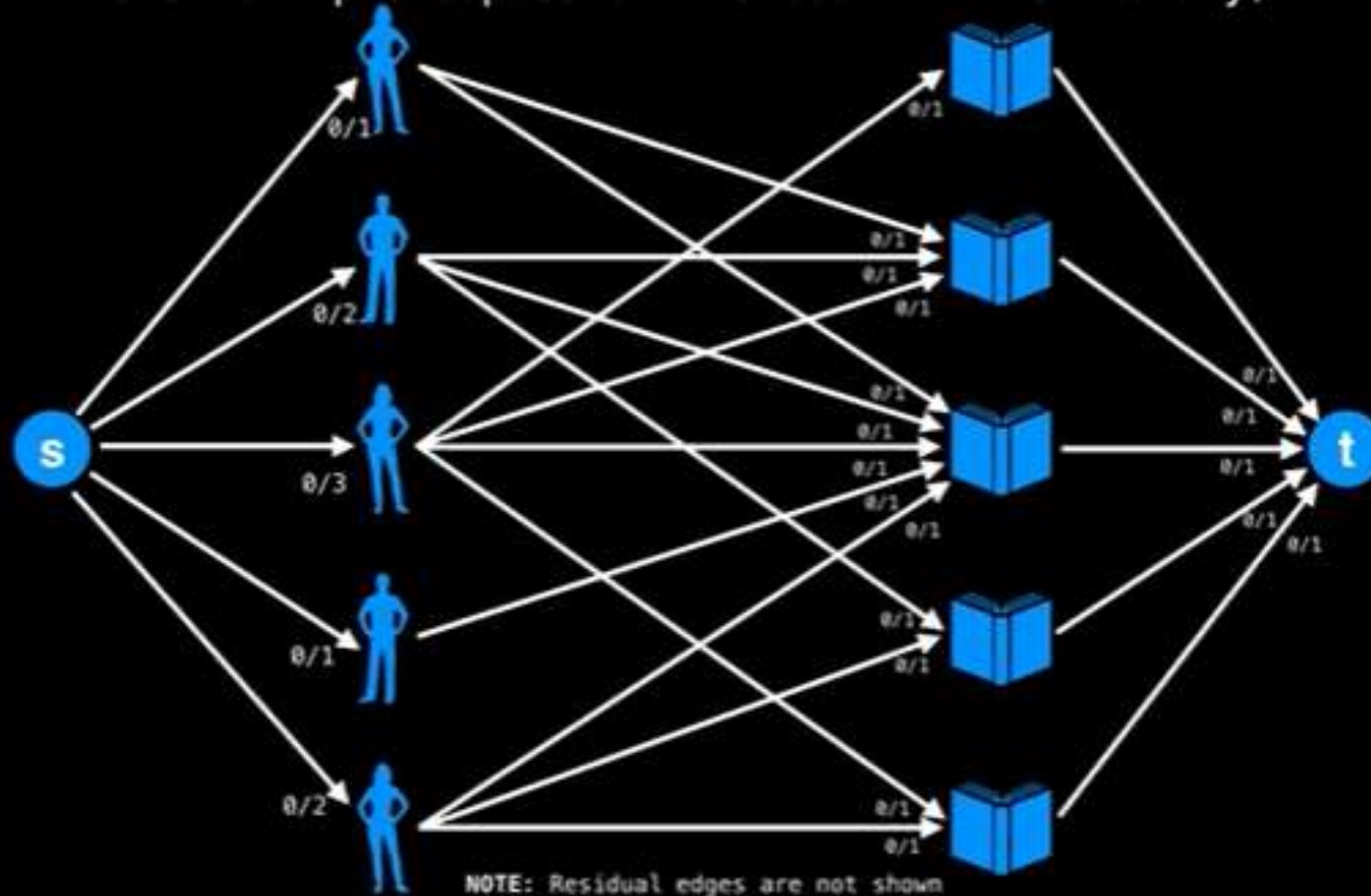


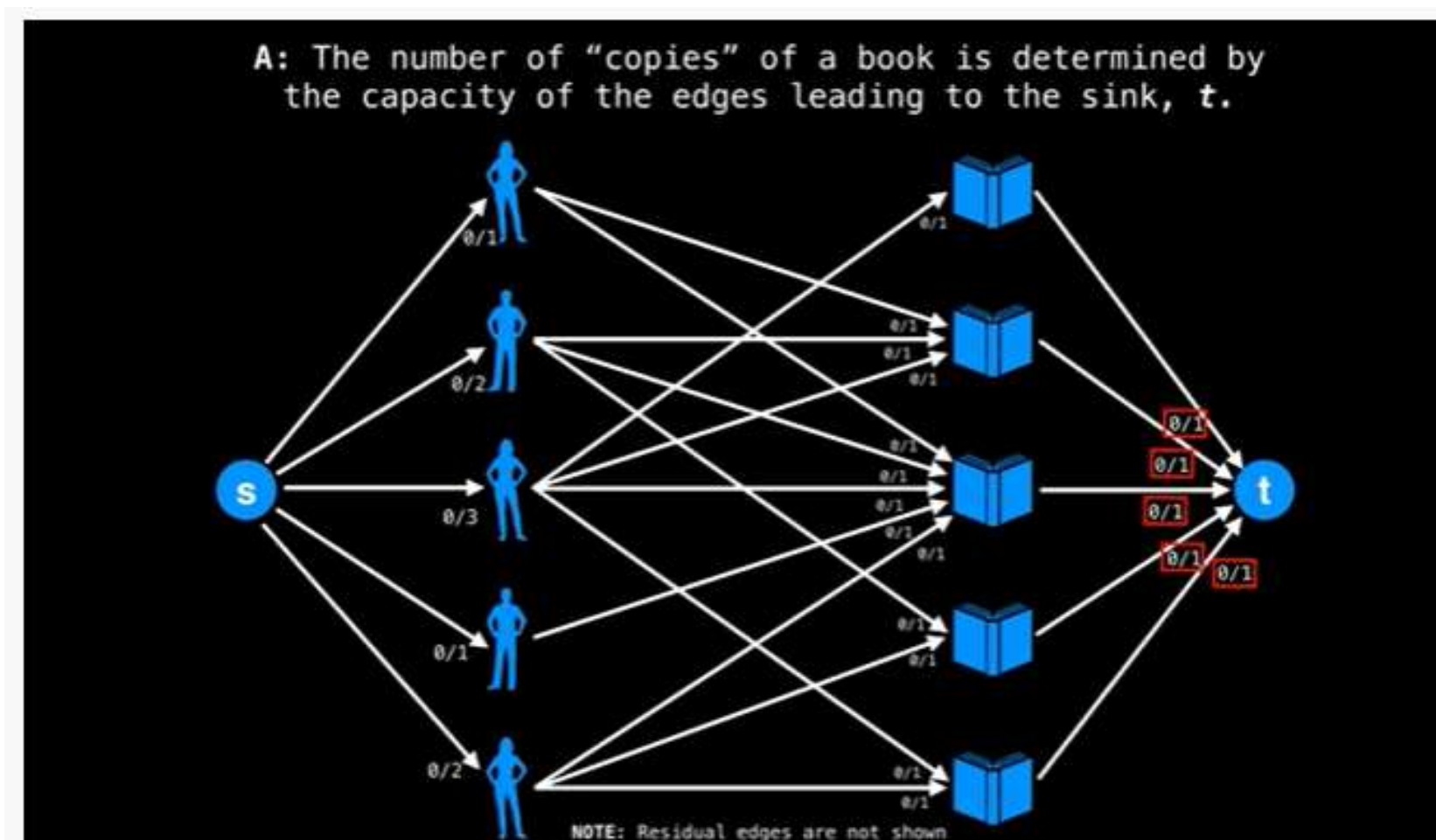


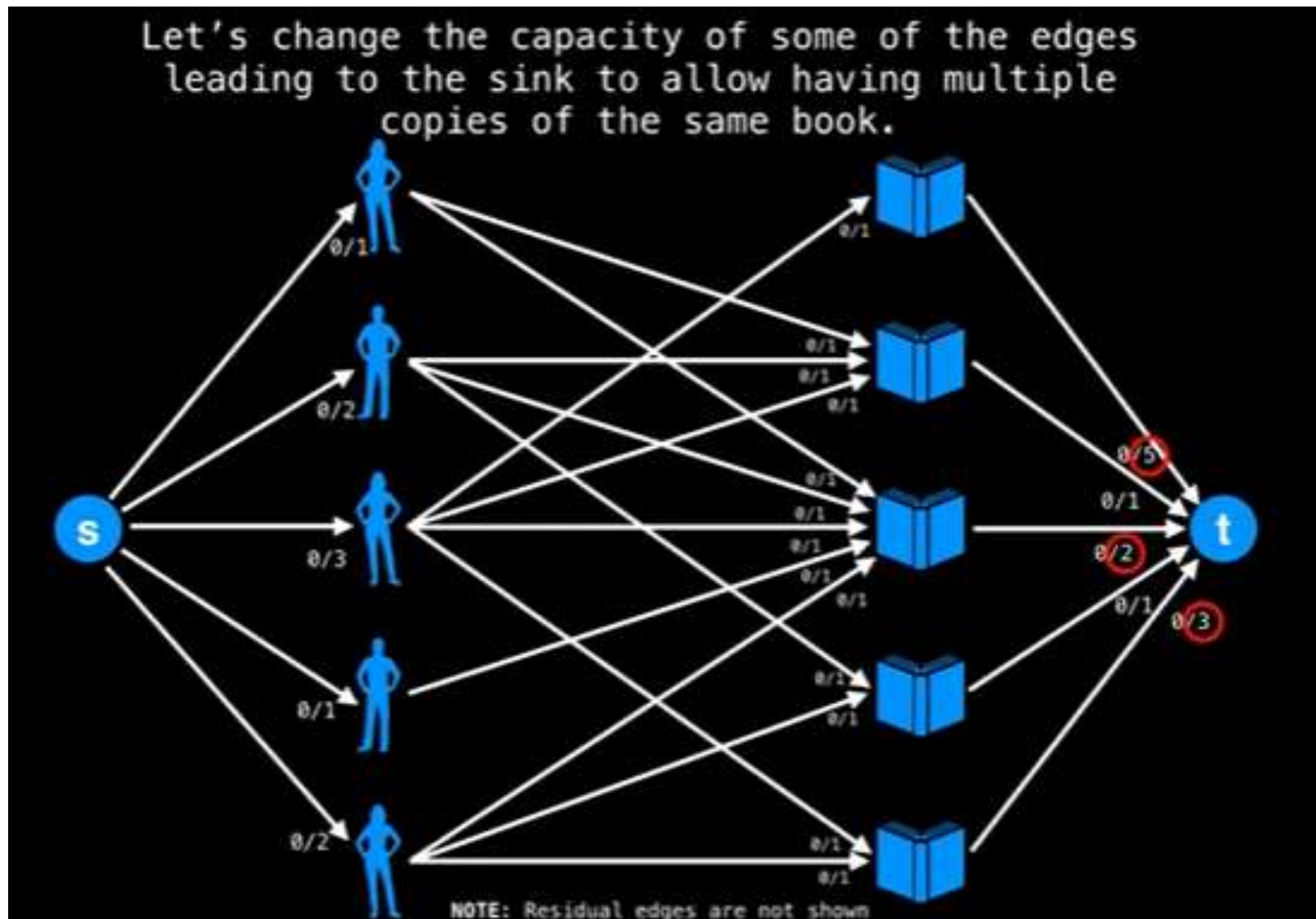


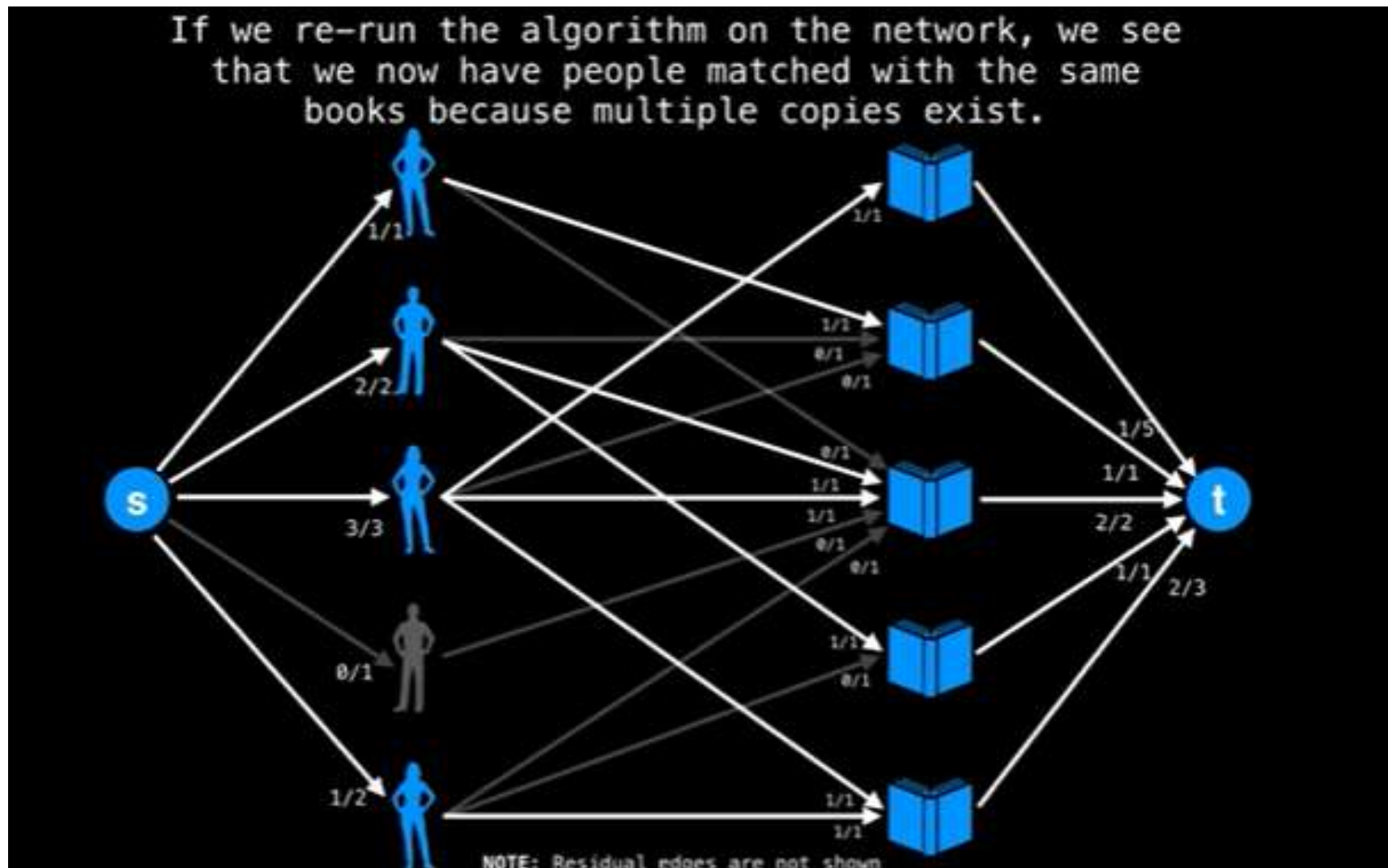


Q: What do we need to change in the flow network to allow a book to be selected multiple times (e.g there are multiple copies of the book in the library)?

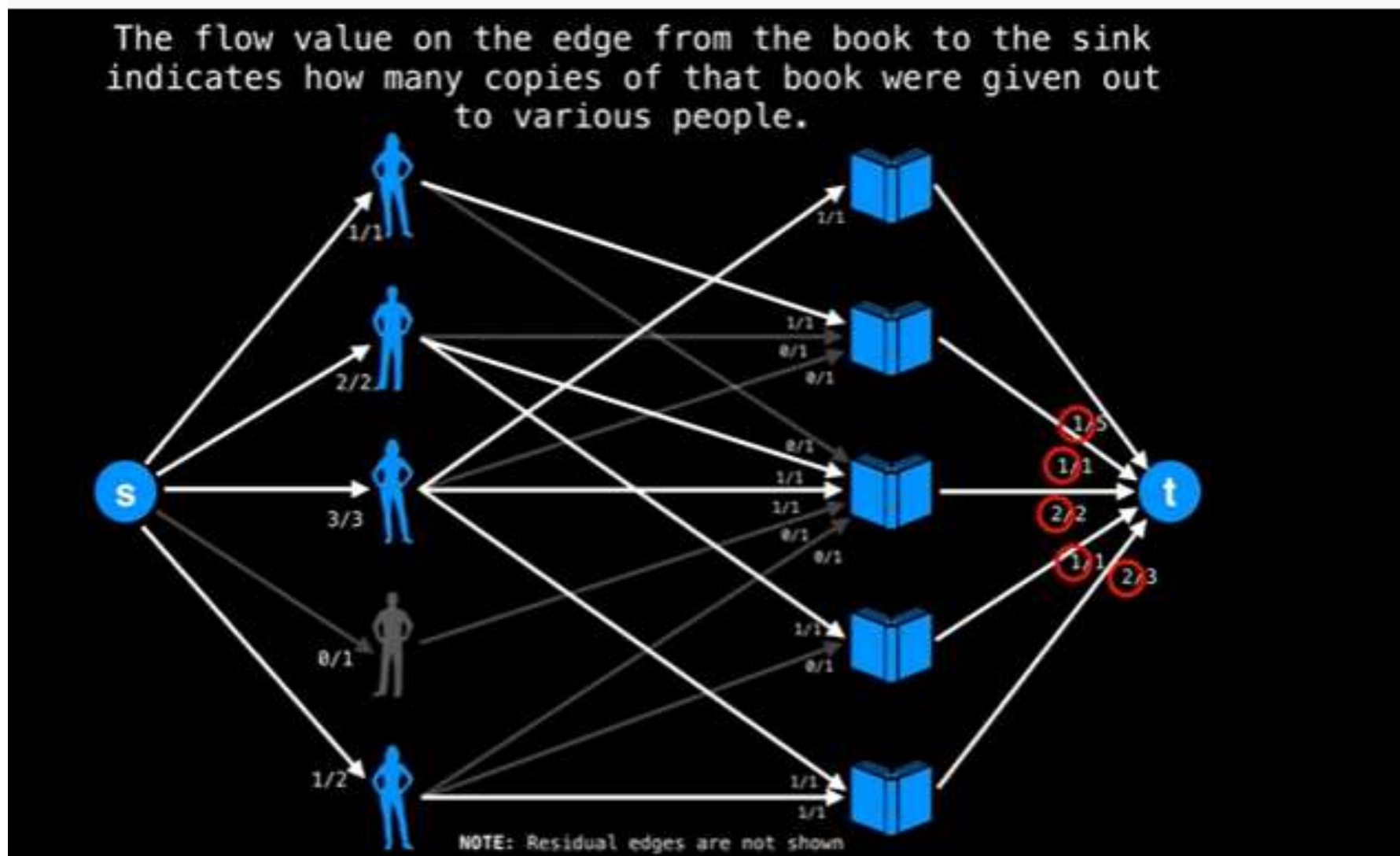




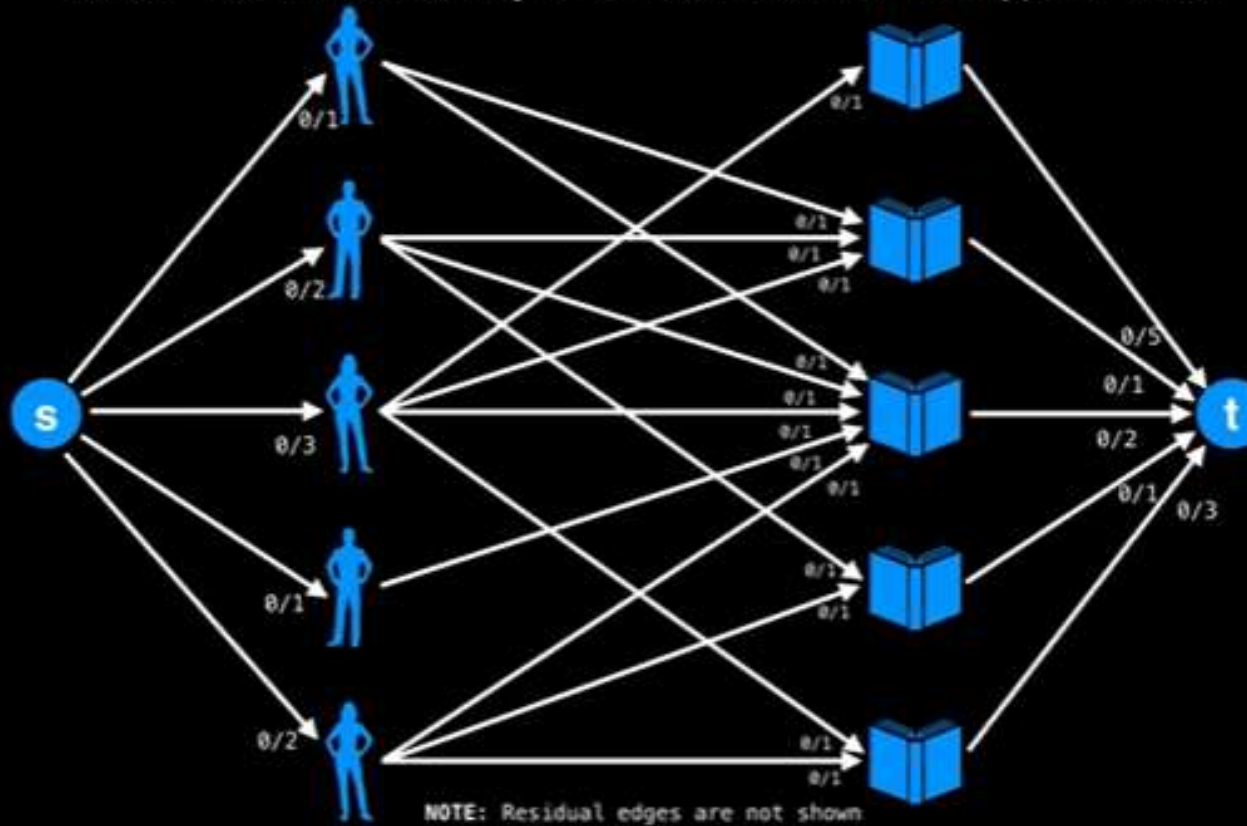








Q: Currently, each person is only allowed to pick up one copy of each book (even though there are multiple copies of each book). How do we modify the flow network to support this?



A: Modify the edge capacity between a person and a book to allow that person to pick up multiple copies of that book.

