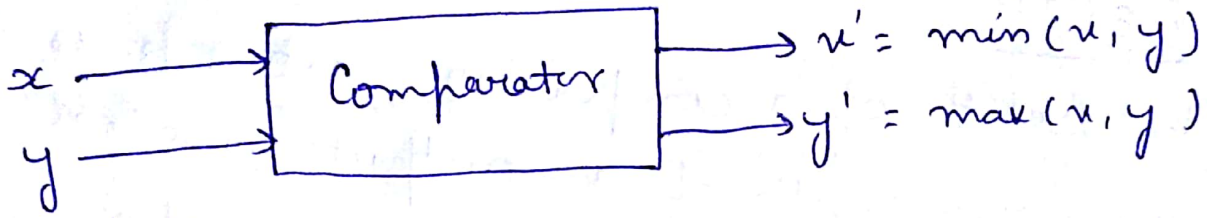
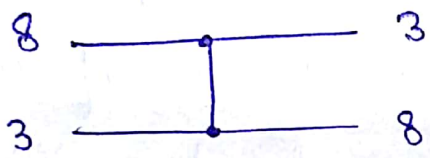
UNIT-IV

Comparison Network: It is composed of wires and comparator.

Comparator: It is a device with 2 inputs 'x' & 'y' & two outputs x' & y'. And it is used to compare two bits with the smaller input value appearing on the top output & the larger input value appearing on the bottom output.



Eg:



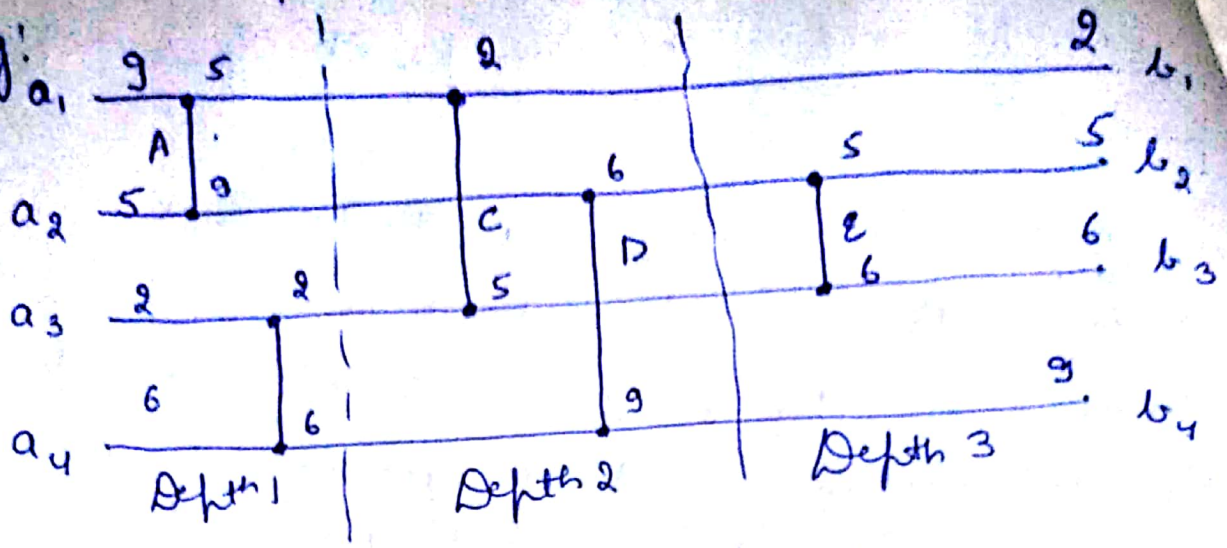
* **Wires**: • A wire transmit a value from place to place. A comparison network contains 'n' input wires $a_1, a_2, a_3 \dots a_n$ through which the values to be sorted enter the network & 'n' output wires $b_1, b_2 \dots b_n$ which produce the results computed by the network.

• A comparison network of 'n' inputs as a collection of 'n' horizontal lines with comparators stretched vertically.

• A line does not represent a single wire but rather a sequence of distinct wires connecting

various comparators.

Eg:



* DEPTH:

- The Depth of a comparison network is the maximum depth of an output wire or equivalently the maximum depth of a comparator.
- The depth of the network therefore equals the time for the n/w to produce values at all of its output wires.
- In the above, depth is 3 as the O/P appears twice on output wires to receive their values once the input receives their.

* Requirement of Comparison N/w:

The main req. for inter connecting comparators is that the graph of interconnections must be acyclic i.e. if we trace a path from the O/P of a given comparator to I/P of another, the path we trace must never cycle back on itself & through the same comparator twice.

Comparison N/w is of 2 types:

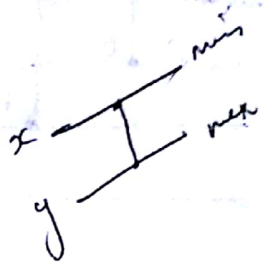
Comparison Network
- Sorting N/w (Bitonic SN)
- Merging N/w
  - Sorting

* BSN → It is a comparison network that is used to sort bitonic input sequence i.e. a sequence that monotonically ↑ses & then monotonically ↓ses

Eg → (1, 4, 6, 8, 3, 2)

(6, 9, 4, 2, 3, 5)

(8, 5, 3, 4, 9)



→ The zero-one sequences have the form $0^i 1^j 0^k$ or $1^i 0^j 1^k$ for some $i, j, k \geq 0$.

→ Any sequence that is either monotonically ↑ng or monotonically ↓ng is also bitonic

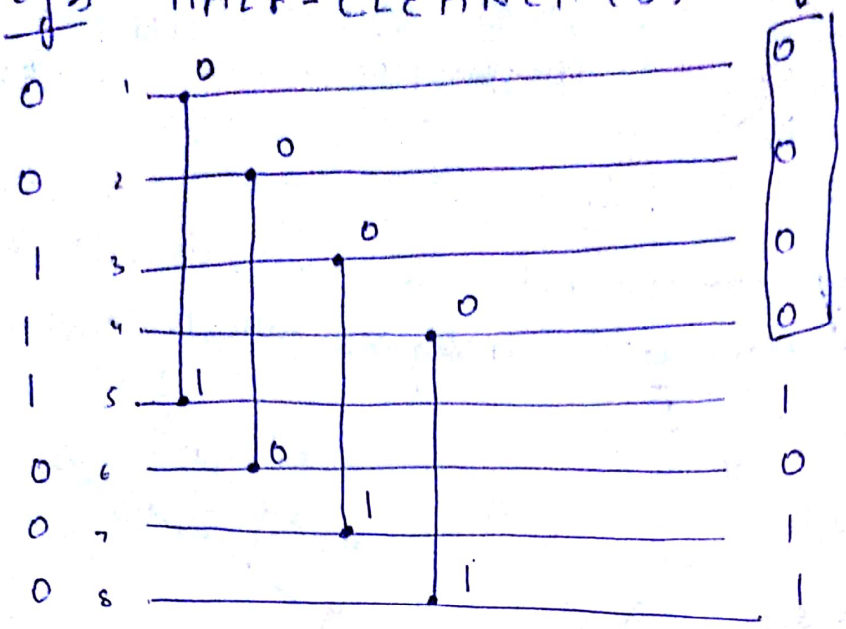
* CONSTRUCTION OF BITONIC SORTER →

→ It is composed of several stages, each stage is known as half cleaner.

→ Each half cleaner is a comparison network in which input line 'i' is compared with line $i + n/2$ for $i = 1, 2, \ldots n/2$ (where 'n' is even)

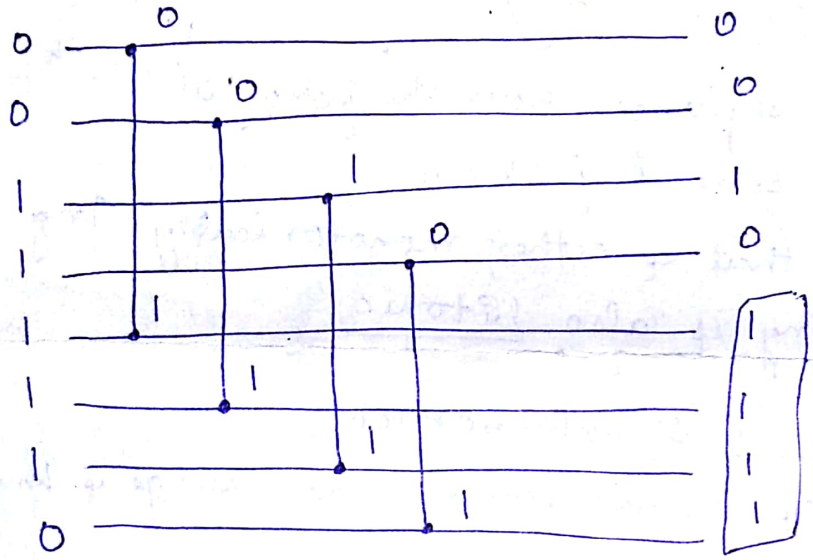
→ When a bitonic sequence of 0's & 1's is applied as input to a half-cleaner, the half cleaner produces an O/P sequence in which smaller values are at top half, larger values in the bottom half & both the halves are bitonic & atleast one of the halves is clean i.e. consisting of either all 0's or all 1's. Hence it is named as HALF-CLEANER.

Eg:- HALF-CLEANER (8)



i	i+n/2
1	5
2	6
3	7
4	8

n = 8



* BITONIC SORTER :→ By recursively combining half cleaners, Bitonic sorter can be build which is a network that sort bitonic sequences.

→ The first stage of Bitonic Sorter [n] consists of HALF-CLEANER [n] which produces 2 bitonic sequences of half the size of original input.

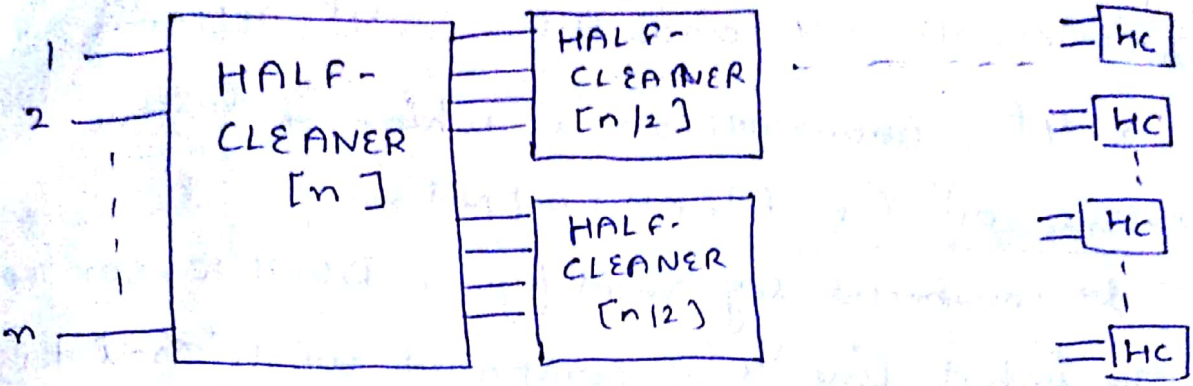
→ Then the two copies of BITONIC SORTER [n/2] are used to sort the two halves recursively.

The depth D(n) of BS[n] is given by:

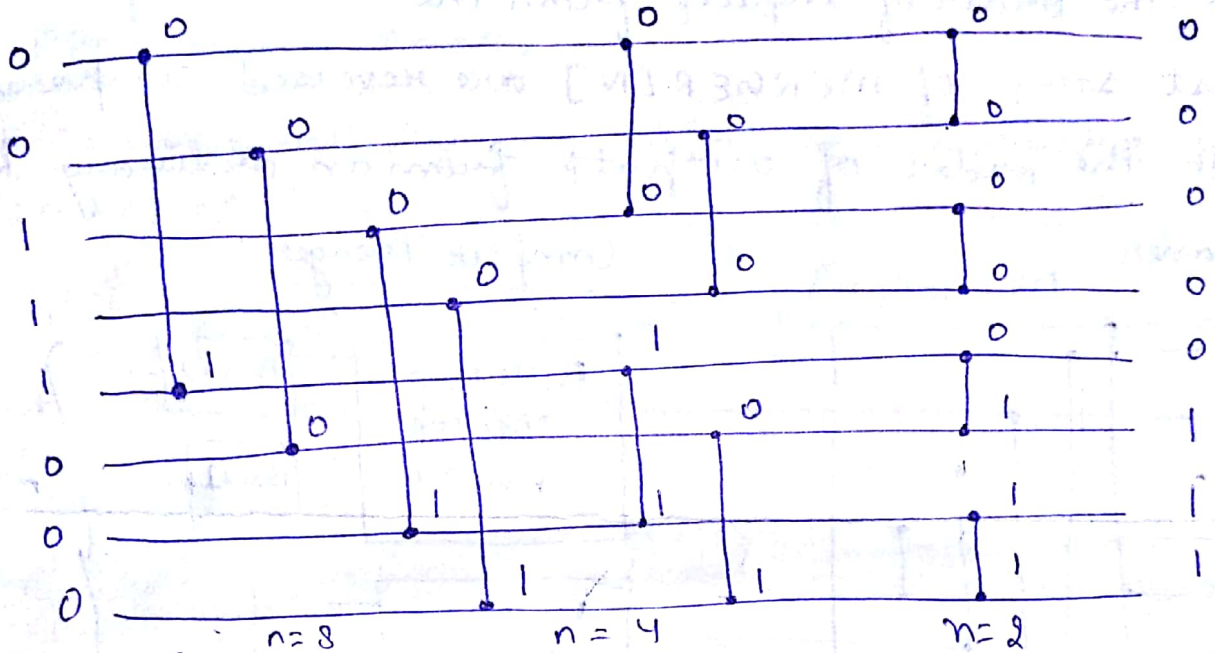
$$D(n) = \begin{cases} 0 & \text{if } n = 1 \\ D(n/2) + 1 & \text{if } n = 2^k \text{ \& } k \geq 1 \end{cases}$$

Soln is $D(n) = \log n$

BLOCK DIAGRAM OF BITONIC SORTING N/W →



Eg → BITONIC SORTER[8] for input 00111000



i	i+n/2
1	5
2	6
3	7
4	8

i	i+n/2
1	3
2	4

i	i+n/2
1	2

* MERGING NETWORK → It is a sorting network that can merge two sorted input sequences into one sorted O/P sequence.

PRINCIPLE → Given two sorted sequences, if we reverse the order of second sequence & then concatenate the two sequences, the resulting sequence is bitonic.

For eg : Consider two sequences:

$X = 00000111$, $Y = 00001111$ $\quad Y^R = 11110000$

$X.Y^R = 0000011111110000$ which is bitonic.

* CONSTRUCTION OF MERGER [N] →

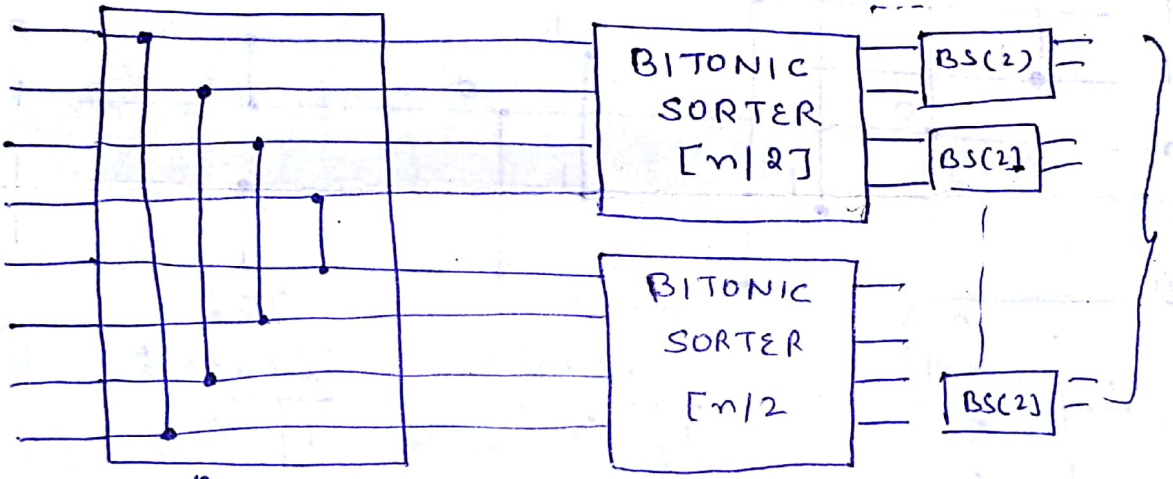
It can be constructed by modifying BITONIC-SORTER [n].

Here, the input line 'i' is compared with 'n-i+1'.

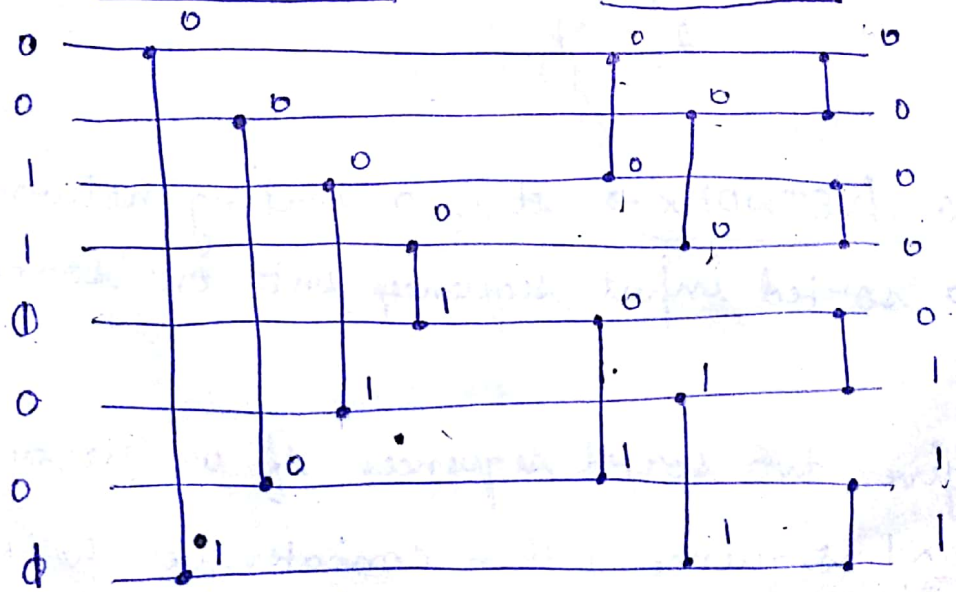
Also, the order of inputs from the bottom of the first stage of MERGER [N] are reversed compared with the order of outputs from an ordinary half cleaner.

MERGER [n]

Complete Merger :



Eg →



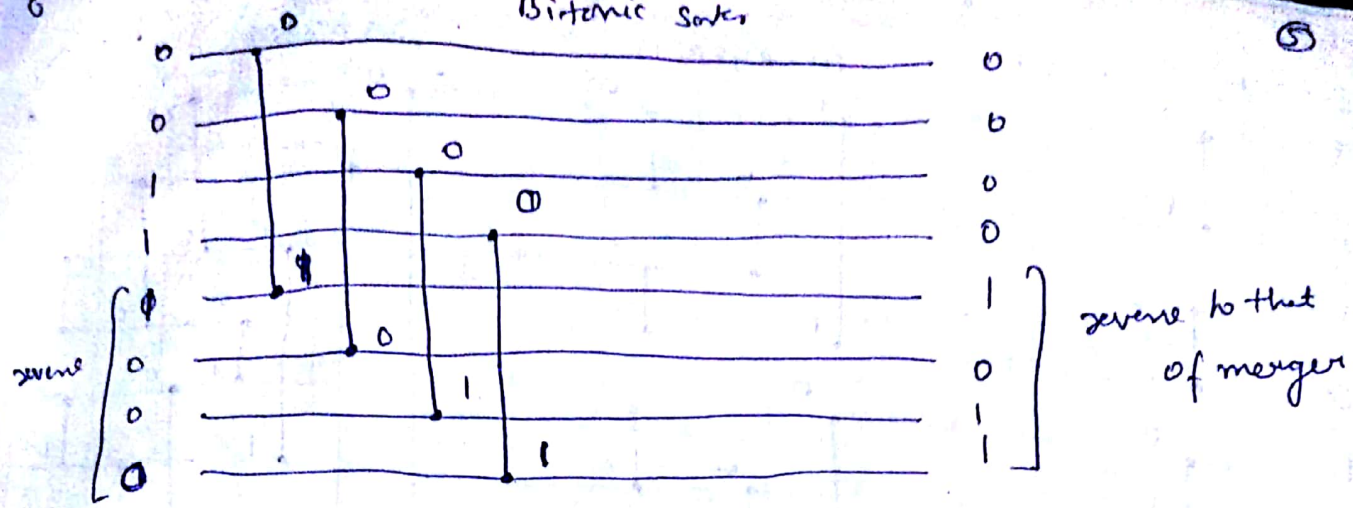
$X = 0011$

$Y = 1000$

$Y^R = 0001$

i	n-i+1
1	8
2	7
3	6
4	5

Bitonic Sorter



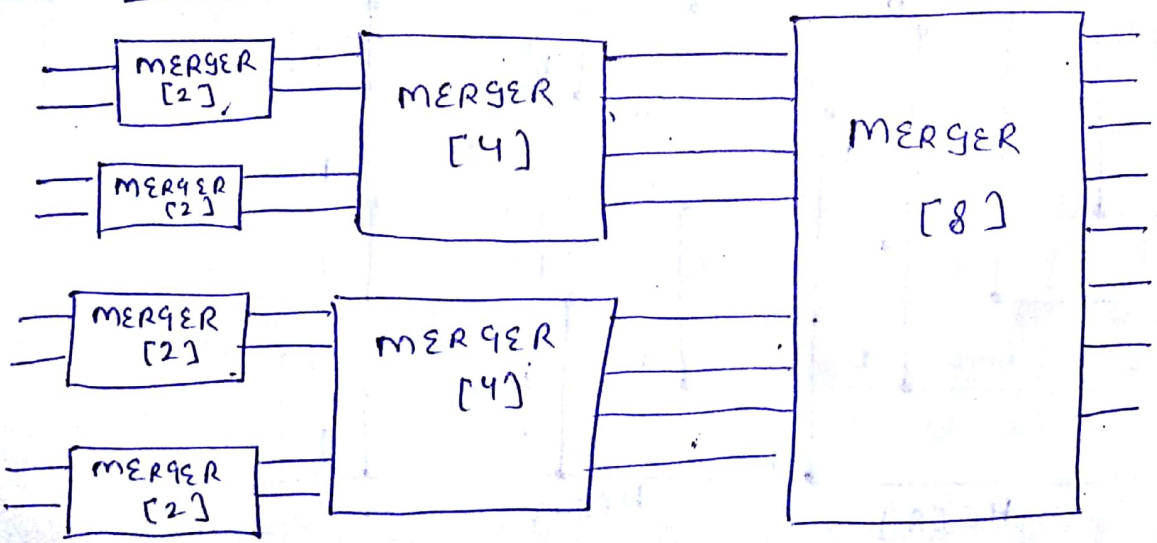
* SORTING NETWORK →

The sorting network SORTER[n] is constructed using MERGER[n]. The first stage of SORTER[n] consists of n/2 copies of MERGER[2] that work in parallel. Then, the second stage consists of n/4 copies of Merger[4]. Hence, stage 'k' consists of $n/2^k$ copies of Merger[2^k]. The depth of sorter is given by the recurrence:

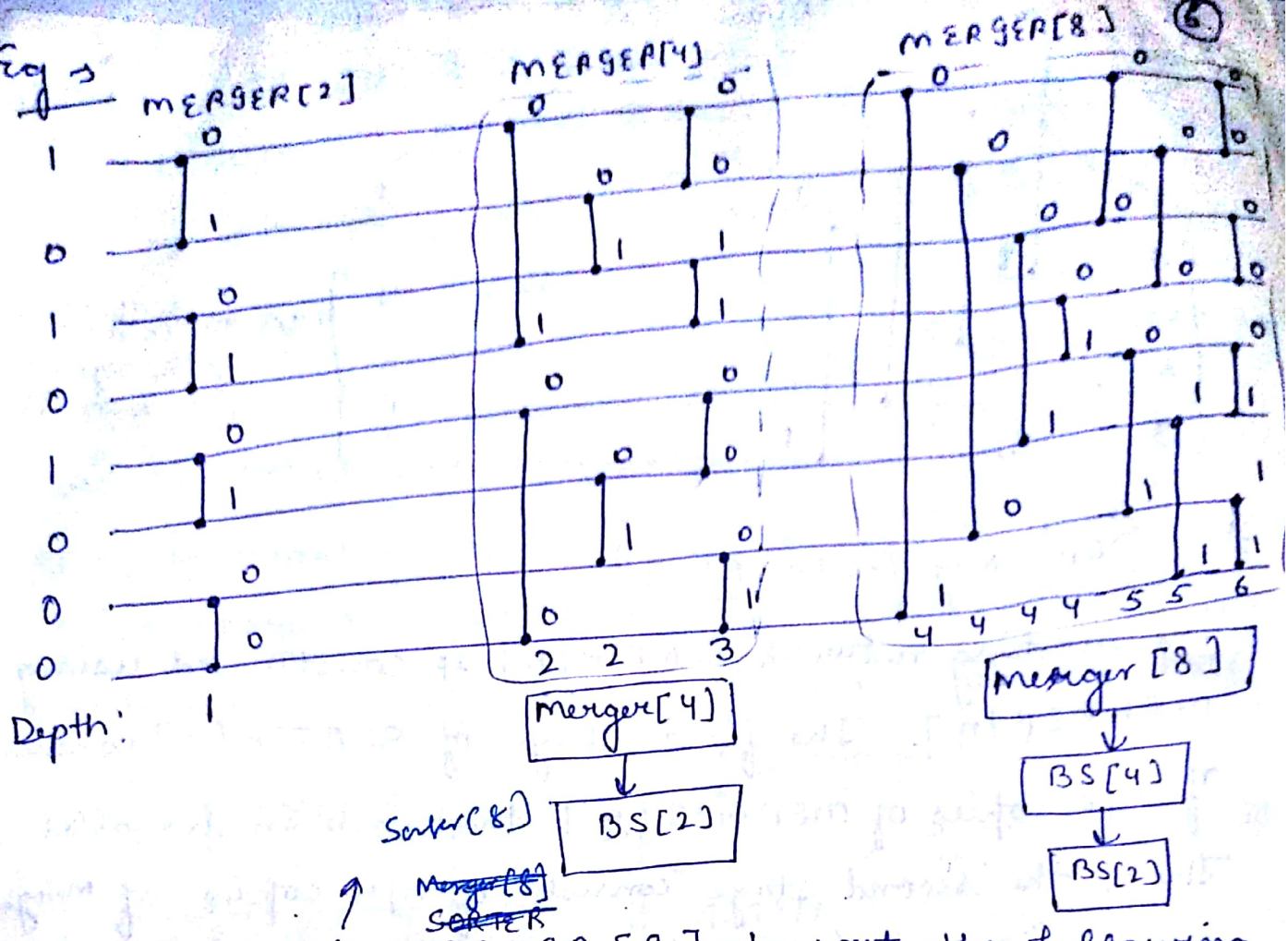
$$D(n) = \begin{cases} 0 & \text{if } n = 1 \\ D\left(\frac{n}{2}\right) + \log n & \text{if } n = 2^k \text{ & } k \geq 1 \end{cases}$$

& we can sort 'n' numbers in parallel in $O(\log^2 n)$ time.

* BLOCK DIAGRAM OF SORTING NETWORK →



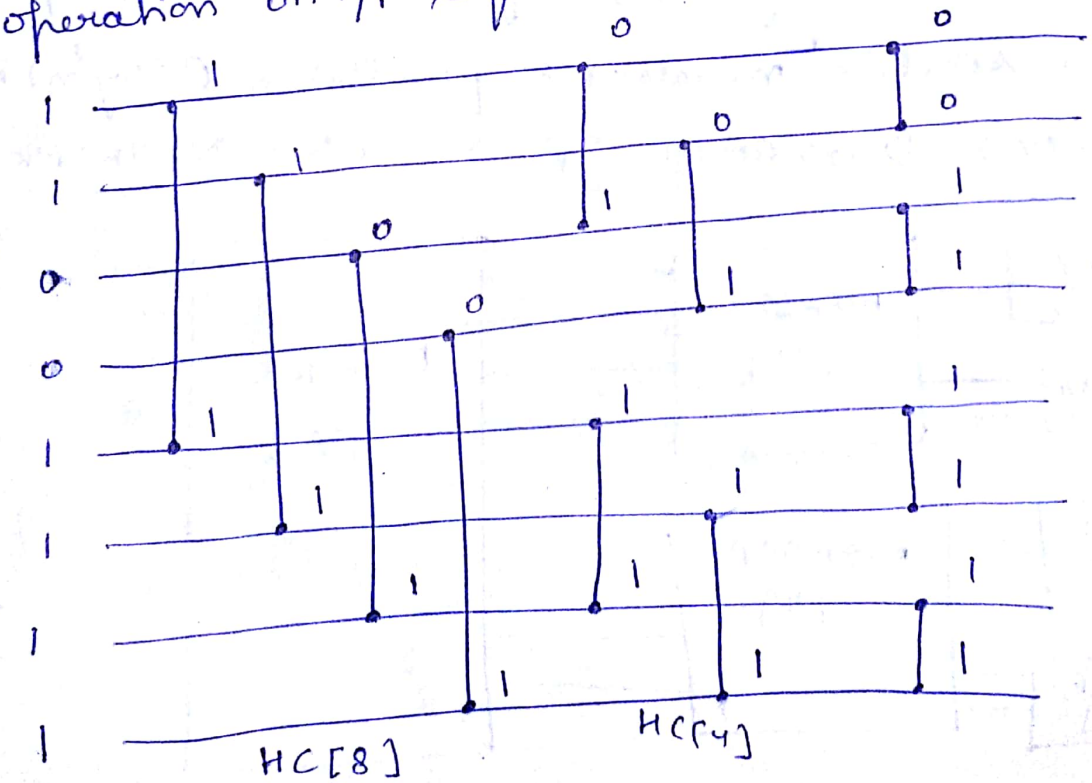
Eg →

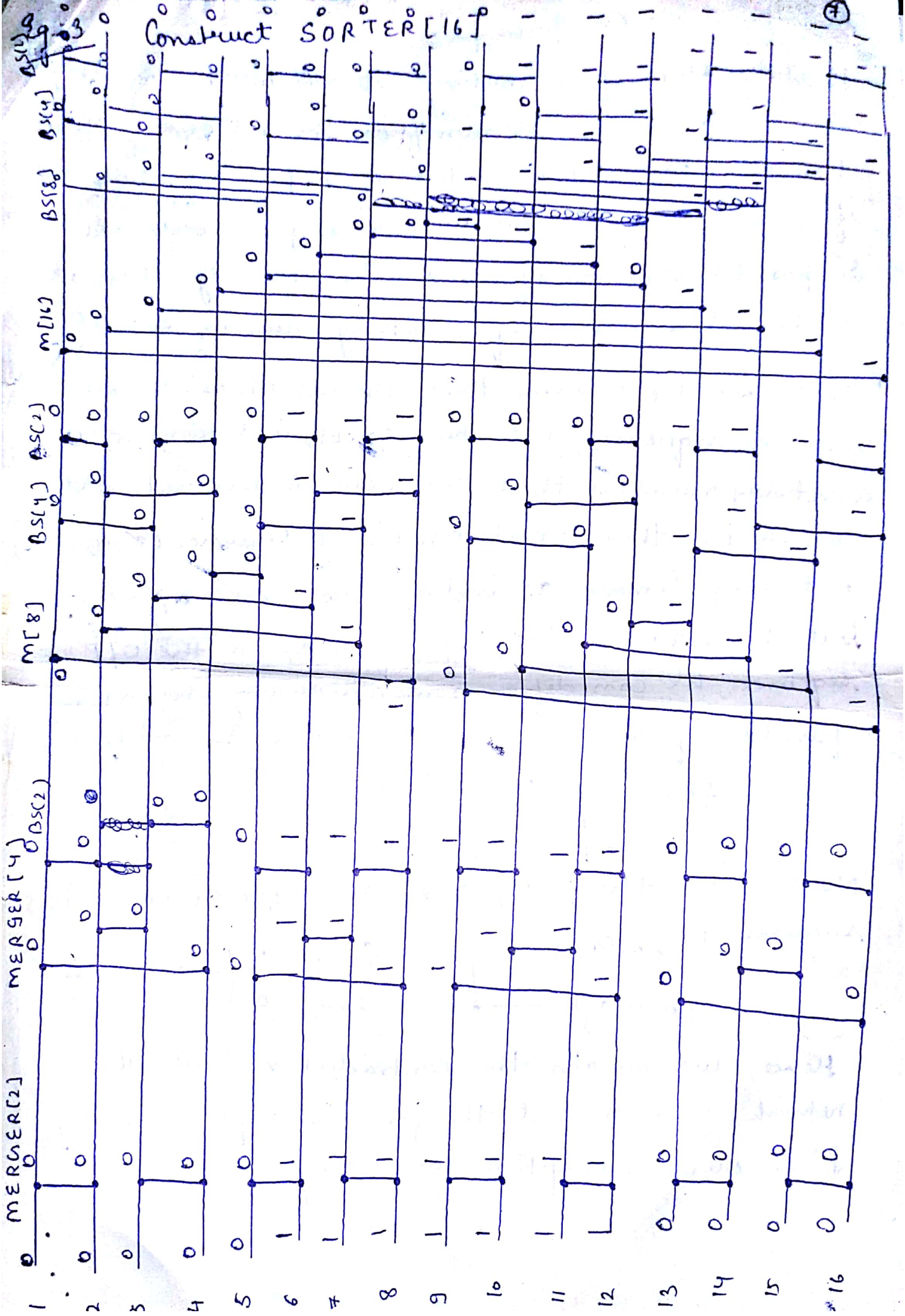


Eg-1 Construct ~~MERGER~~ [8] to sort the following two I/P sequences $X = 1010$, $Y = 0001$, $Y^R = 1000$

$X.Y^R = 10101000$

Eg-2 Construct BITONIC SORTER [8] & show its operation on I/P sequence 11001111.





* The zero-one principle:

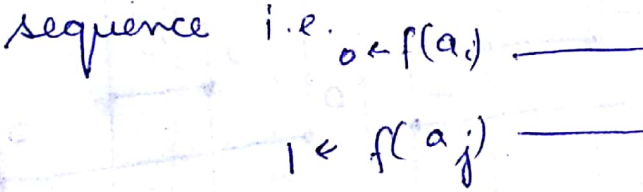
It states that if a sorting network works correctly when each input is drawn from the set $\{0, 1\}$, then it works correctly on arbitrary input numbers. i.e. if a comparison network with 'n' input sorts all 2^n possible sequences of 0's and 1's correctly, then it sorts all sequences of arbitrary numbers correctly.

Proof: Let us assume that the network sorts all zero-one sequences, but there exists a sequence of arbitrary numbers that the network does not sort correctly i.e. there exists an input sequence $(a_1, a_2 \dots a_n)$ containing elements a_i and a_j such that $a_i < a_j$, but the network places a_j before a_i in the O/P sequence. We can define a monotonically increasing function 'f' as:

$a_1 < a_2 < a_3 \dots a_i < a_j \dots$

$$f(x) = \begin{cases} 0 & \text{if } x \le a_i, \\ 1 & \text{if } x > a_i \end{cases}$$

Now, since the n/w places a_j before a_i in input sequence i.e.



Hence, we obtain the contradiction that the network fails to sort the zero-one sequence correctly.

$\therefore$ our assumption was wrong.

FLOW NETWORKS →

A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$. Every vertex lies on some path from source to sink i.e. $s \rightsquigarrow v \rightsquigarrow t$.

Let $G = (V, E)$ be a flow network with a capacity function 'c'. Let 's' be the source of network & 't' be the sink. A flow in 'G' is a real-valued function $f : V \times V \to R$ that satisfies the following three properties:

a) **Capacity Constraint** → For all $(u, v \in V)$, $f(u, v) \leq c(u, v)$
Flow from one vertex to another must not exceed the given capacity.

b) **Skew Symmetry** → For all $u, v \in V$, $f(u, v) = -f(v, u)$
It says that flow from a vertex 'u' to 'v' is the -ve of the flow in the reverse direction.

c) **Flow Conservation** → For all $u \in V - \{s, t\}$

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

Total +ve flow entering a vertex other than source or sink must equal the total +ve flow leaving that vertex.

(It says that the total flow out of a vertex other than source or sink is

In **MAXIMUM FLOW PROBLEM**, from a flow network 'G' with source 's' & sink 't', we need to find a flow of maximum value.

* **Some Properties of Flow** →

Let $G = (V, E)$ be a flow network & let 'f' be a flow in 'G'

1) For all $X \subseteq V$, we have $f(X, X) = 0$

2) For all $X, Y \subseteq V$, $f(X, Y) = -f(Y, X)$

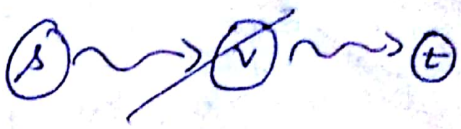
3) For all $X, Y, Z \subseteq V$ with $X \cap Y = \phi$

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$$

LEMMA-1 Prove that the total flow entering leaving the source is equal to the flow entering the sink.

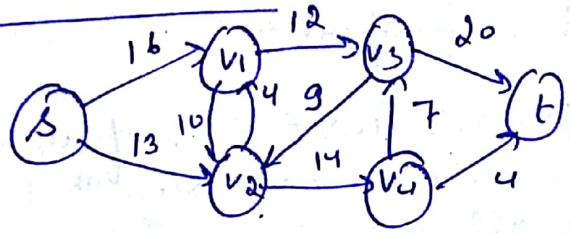
$f(s, V)$



* FORD - FULKERSON METHOD →

The algorithm begins by initializing $f(u,v)=0$ for all $(u,v) \in V$ giving an initial flow of value 0.

Flow Residual Network



Residual N/w : Given a flow n/w 'G' & a flow 'f', the residual n/w consists of edges with capacities that represent the additional flow that can be sent via edges.

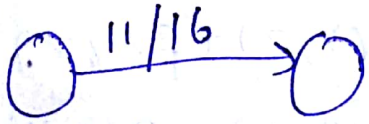
An edge of the flow n/w can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge.

The residual capacity can be defined as;

$$c_f(u,v) = c(u,v) - f(u,v)$$

c_f ↑ Residual capacity; c ↑ Total cap.; f ↑ flow

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



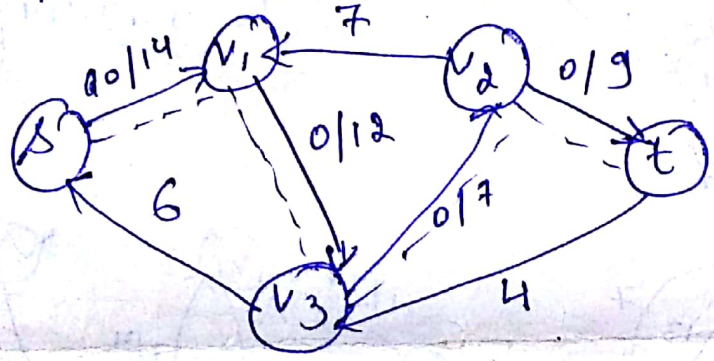
$c_f = 16 - 11 = 5$

Augmenting Paths:

Given a flow n/w $G_1 = (V, E)$ and a flow 'f', an augmenting path 'p' is a path from source 's' to sink 't' in the residual network G_{1f}.

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

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



FORD-FULKERSON METHOD: This method is iterative & we start with $f(u,v) = 0$ for all $(u,v) \in V$, giving an initial flow value = 0. At each iteration, flow value is increased by finding an augmenting path from 's' to 't'. This process is repeated until no augmenting path is left. The corresponding flow n/w yields max. flow.

Algo ford fulkerson (G, s, t)

1. for each edge $(u,v) \in E[G]$
2. $f[u,v] := 0$
3. while there exists a path 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. $f[u,v] := f[u,v] + c_f(p)$

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

Here, 'G' is the flow network consisting of 'E' edges, 'f' is the flow, G_f is the residual network, 'p' is an augmenting path, c_f is the residual capacity.

The running time of the algorithm is

$$O(E|f^*|)$$

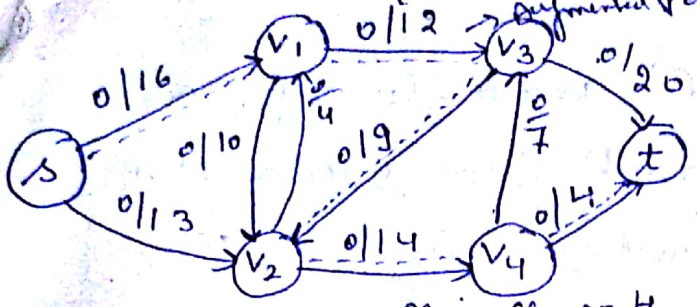
where 'E': The no. of edges.

f^*: Maximum flow in the network.

* Application of Ford-Fulkerson Method:

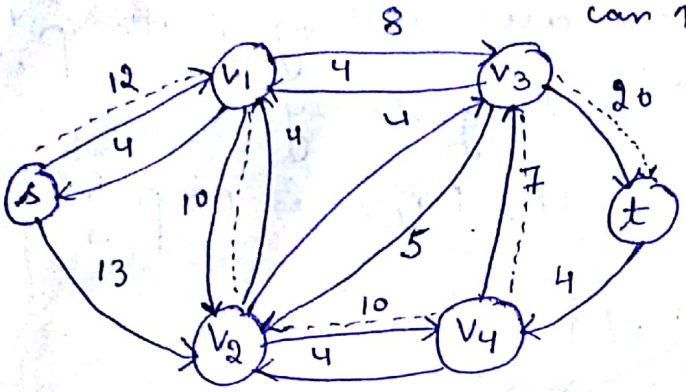
- Maximum Bipartite Matching
- Max-flow Min-Cut Theorem

FORD-FULKERSON METHOD → Augmented Path

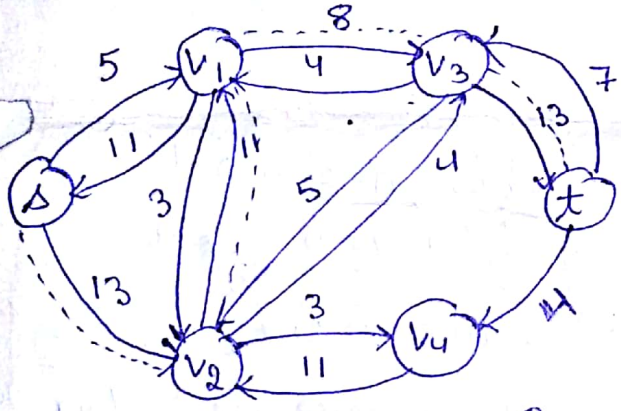


Min flow = 4

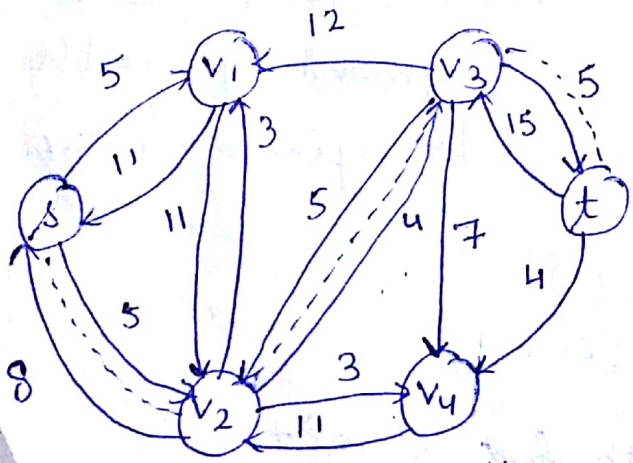
Residual N/w (how much flow we can use)



Min flow = 7

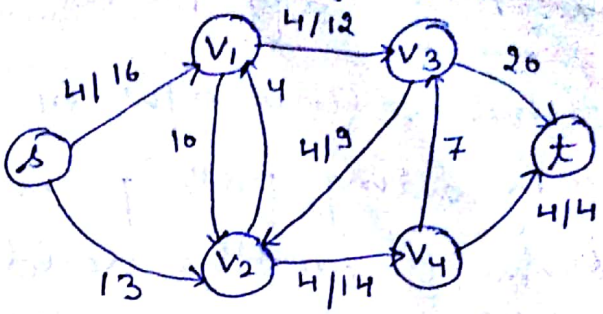


Min flow = 8

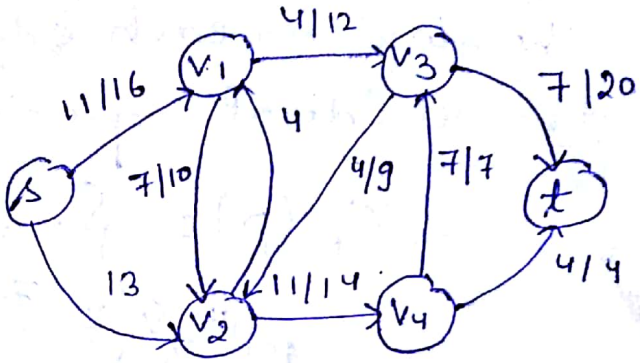


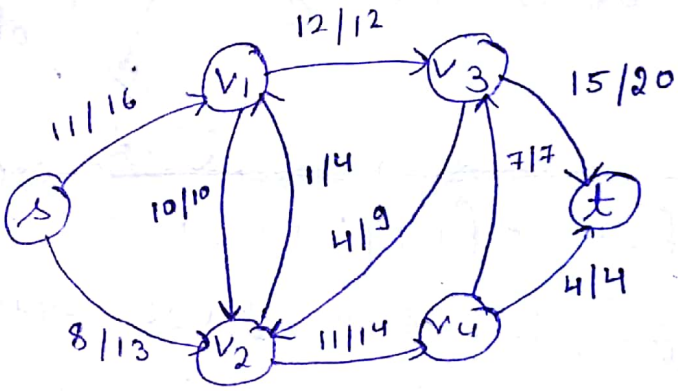
Min flow = 4

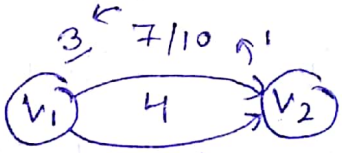
Residual capacity = Capacity - Flow



Flow network.

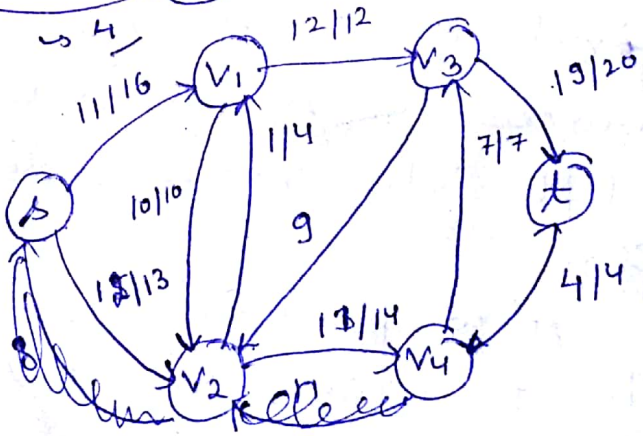






$V_2 \xrightarrow{8} V_1$

$\Rightarrow 4$



We don't add '4' b/c edge is opposite,

∴ we take as -4 ∑ $4-4=0$

Maximum Bipartite Matching:

Given an undirected graph $G = (V, E)$, a matching is a subset of edges $M \subseteq E$ such that for all vertices, $v \in V$, atmost one edge of 'M' is incident on 'v'.

A vertex, $v \in V$ is matched by the matching 'M' if some edge in 'M' is incident on 'v', otherwise 'v' is said to be unmatched.

A maximum matching is a matching of maximum cardinality.

The bipartite graphs are those in which vertex set can be partitioned into $V = L \cup R$ where L and R are two disjoint set of vertices & all edges go between L and R.

* To find Maximum Bipartite Matching:

Let there be a bipartite graph $G = (V, E)$. A corresponding flow network $G' = (V', E')$ can be defined for the given graph 'G'.

Let source 's' and sink 't' be the two new vertices not in 'V' such that:

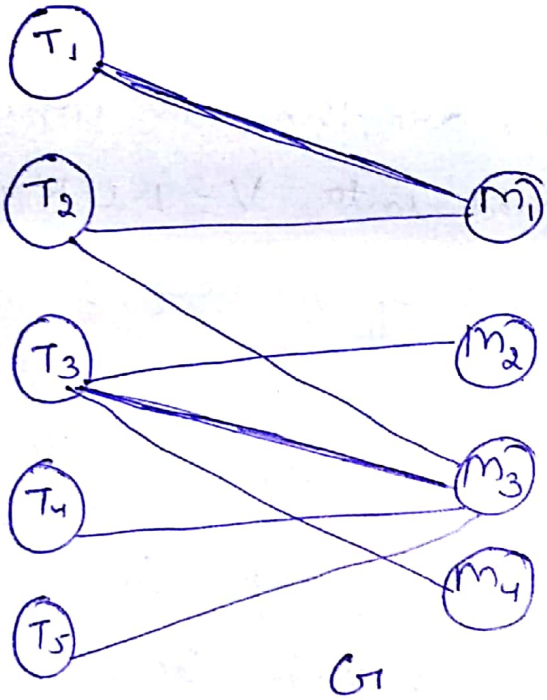
$$V' = V \cup \{s, t\}$$

The set of edges E' for the flow network G' can be defined as:

$$E' = \{(s,u) : u \in L\} \cup \{(u,v) : (u,v) \in E\} \cup \{(v,t) : v \in R\}$$

Union

Eg: Consider the following set of tasks and machines. The edges indicate that a particular task can be performed by a particular machine. Also, 1 machine can perform atmost 1 task.

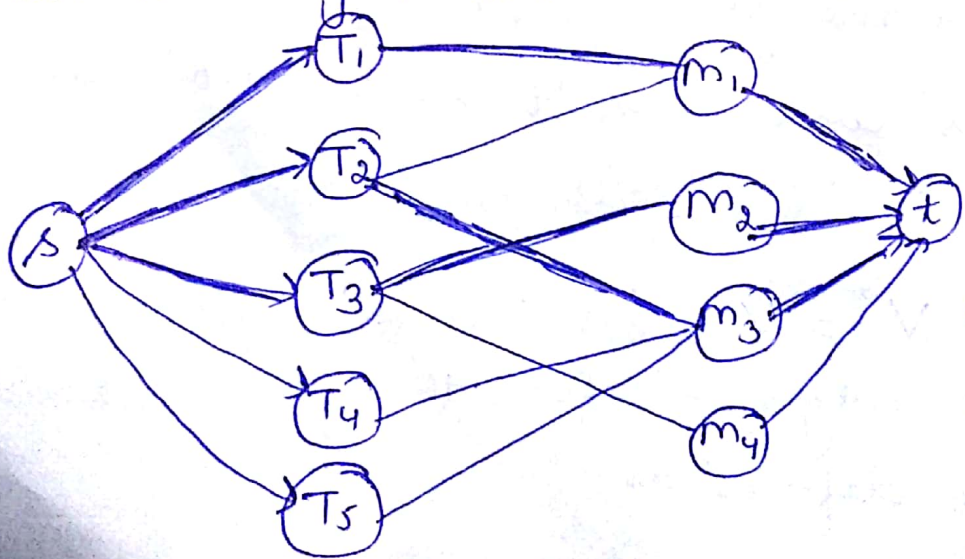


G

BIPARTITE MATCHING,

Cardinality = |2|

Constructing Flow Network G':



MAXIMUM BIPARTITE MATCHING,

Cardinality = |3|

In optimization theory, the **max-flow min-cut theorem** states that in a flow network, the maximum amount of flow passing from the *source* to the *sink* is equal to the minimum capacity that, when removed in a specific way from the network, causes the situation that no flow can pass from the source to the sink.

Definitions and Statement

Let $N = (V, E)$ be a network (directed graph) with s and t being the source and the sink of N respectively.

Maximum Flow

Definition. The **capacity** of an edge is a mapping $c : E \to \mathbf{R}^+$, denoted by c_{uv} or $c(u, v)$. It represents the maximum amount of flow that can pass through an edge.
Definition. A **flow** is a mapping $f : E \to \mathbf{R}^+$, denoted by f_{uv} or $f(u, v)$, subject to the following two constraints:

1. Capacity Constraint:
$\forall (u, v) \in E : \quad f_{uv} \leq c_{uv}$
2. Conservation of Flows:
$\forall v \in V \setminus \{s, t\} : \quad \sum_{\{u:(u,v)\in E\}} f_{uv} = \sum_{\{u:(v,u)\in E\}} f_{vu}.$

Definition. The **value of flow** is defined by

$|f| = \sum_{v \in V} f_{sv}$, where s is the source of N. It represents the amount of flow passing from the source to the sink.
Maximum Flow Problem. Maximize $|f|$, that is, to route as much flow as possible from s to t.

Minimum Cut: **Definition.** An **s-t cut** $C = (S, T)$ is a partition of V such that $s \in S$ and $t \in T$. The **cut-set** of C is the set

$$\{(u, v) \in E \ : \ u \in S, v \in T\}.$$

Note that if the edges in the cut-set of C are removed, $|f| = 0$.

Definition. The capacity of an *s-t cut* is defined by

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

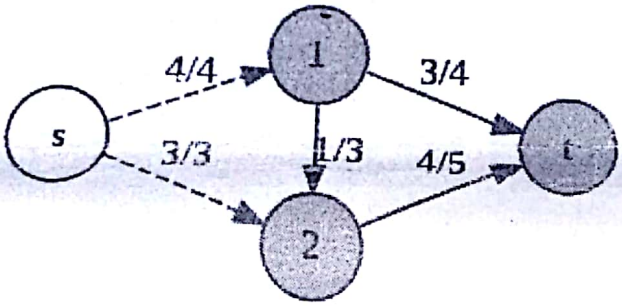
Minimum s-t Cut Problem. Minimize $c(S, T)$, that is, to determine S and T such that the capacity of the *S-T cut* is minimal.

Statement

Max-Flow Min-Cut Theorem. The maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts.

Linear program formulation

Example



A network with the value of flow equal to the capacity of an s-t cut

The figure on the right is a network having a value of flow of 7. The vertex in white and the vertices in grey form the subsets S and T of an s-t cut, whose cut-set contains the dashed edges. Since the capacity of the s-t cut is 7, which equals to the value of flow, the max-flow min-cut theorem tells us that the value of flow and the capacity of the s-t cut are both optimal in this network.

Application

Generalized max-flow min-cut theorem : In addition to edge capacity, consider there is capacity at each vertex, that is, a mapping $c : V \rightarrow \mathbf{R}^+$, denoted by $c(v)$, such that the flow f has to satisfy not only the capacity constraint and the conservation of flows, but also the vertex capacity constraint

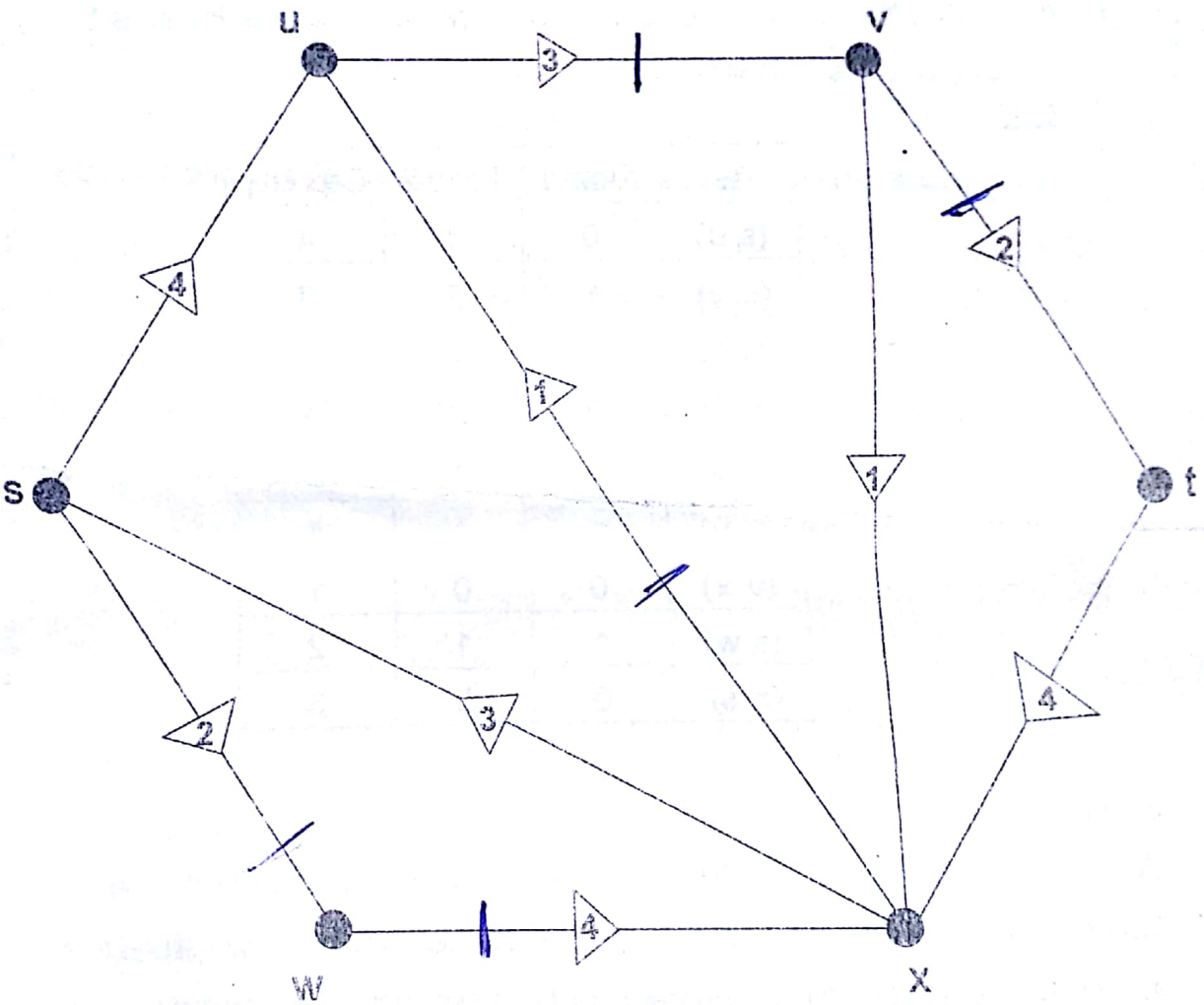
$$\forall v \in V \setminus \{s, t\} : \quad \sum_{i \in V} f_{iv} \leq c(v).$$

In other words, the amount of *flow* passing through a vertex cannot exceed its capacity. Define an *s-t cut* to be the set of vertices and edges such that for any path from s to t, the path contains a member of the cut. In this case, the *capacity of the cut* is the sum the capacity of each edge and vertex in it.

Max Flow Min Cut Algorithm

Example 1.

Find a maximum st-flow and st-minimum cut in the network below starting with a flow of zero in every arc.



We apply the algorithm.

STEP 1.

Now the path swxuvt is a path along which the flow in every arc is less than the capacity in every arc. So we can increase the flow along this path: we can increase the flow by one unit, but no more. So the flow values are now in red in the table below. The value of the flow (net flow leaving s) is 1.

TABLE 1

Arc	Flow 1	Flow 2	Capacity
(s, u)	0	1	4
(u, v)	1	2	3
(v, t)	1	2	2
(t, x)	0	0	4
(w, x)	1	1	4
(x, u)	1	1	1
(v, x)	0	0	1
(s, w)	1	1	2
(x, s)	0	0	3

STEP 2.

We have the red flow in Table 1. Then suvt is a path along which the flow can be increased, by 1 unit, but no more. So the new flow values are shown in the table in blue. The value of the flow (net flow leaving s) is 2.

STEP 3.

We have the blue flow in Table 1. It appears we can't increase the flow so we build the set S for which $(S, \bar{S})$ is a minimum cut. Start with $S = \{s\}$

Since $f(s, w) = 1 < c(s, w) = 2$ we add $w \in S$.
Since $f(w, x) = 1 < c(w, x) = 4$ we add $x \in S$.
Since $f(s, u) = 1 < c(s, u) = 4$ we add $u \in S$.
Since $f(u, v) = 2 < c(u, v) = 3$ we add $v \in S$.

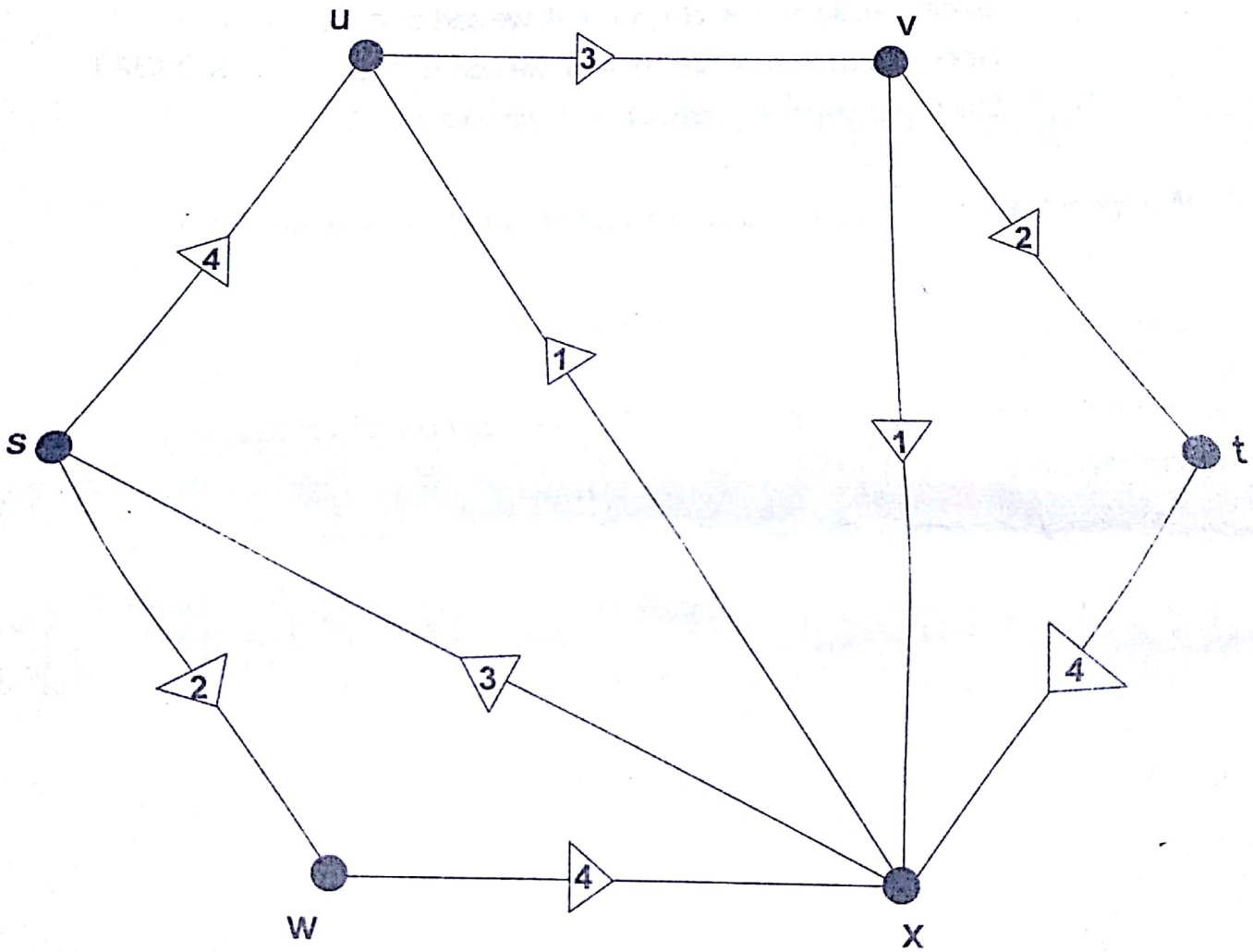
We cannot add t to S, so we have found a min cut. The minimum cut is

$$(S, \bar{S}) = \{(v, t)\}$$

and the capacity of the cut is $c(v, t) = 2$. The blue flow, of value 2, is a maximum flow.

Example 2.

We have the same example, except that you are required to start with the flow of one unit along the cycles suvtxs. Note that the value of this flow is zero, since the net flow leaving s is zero.



STEP 1

We have the flow given below in red. The path sxt has a backward flow of 1: the flow in each arc is positive but backwards towards s along this path. Therefore we can increase the flow to t by reducing the flow in all arcs of that path by one unit, but no less. The new flow is in blue below.

TABLE 2.

Arc	Flow 1	Flow 2	Flow 3	Capacity
(s, u)	1	1	2	4
(u, v)	1	1	2	3
(v, t)	1	1	2	2
(t, x)	1	0	0	4
(w, x)	0	0	0	4
(x, u)	0	0	0	1
(v, x)	0	0	0	1
(s, w)	0	0	0	2
(x, s)	1	0	0	3

STEP 2.

We have the flow in blue. Now along the path suvt the flow can be increased by one unit. The new flow is now in green in the table.

STEP 3.

The green flow seems to be a maximum flow, so let's check that this is the case by building a minimum cut. Start with $S = \{s\}$. Then again

Since $f(s, w) = 0 < c(s, w) = 2$ we add $w \in S$.
Since $f(w, x) = 0 < c(w, x) = 4$ we add $x \in S$.
Since $f(s, u) = 2 < c(s, u) = 4$ we add $u \in S$.
Since $f(u, v) = 2 < c(u, v) = 3$ we add $v \in S$.

We cannot add t to S, so we have found a min cut. The minimum cut is

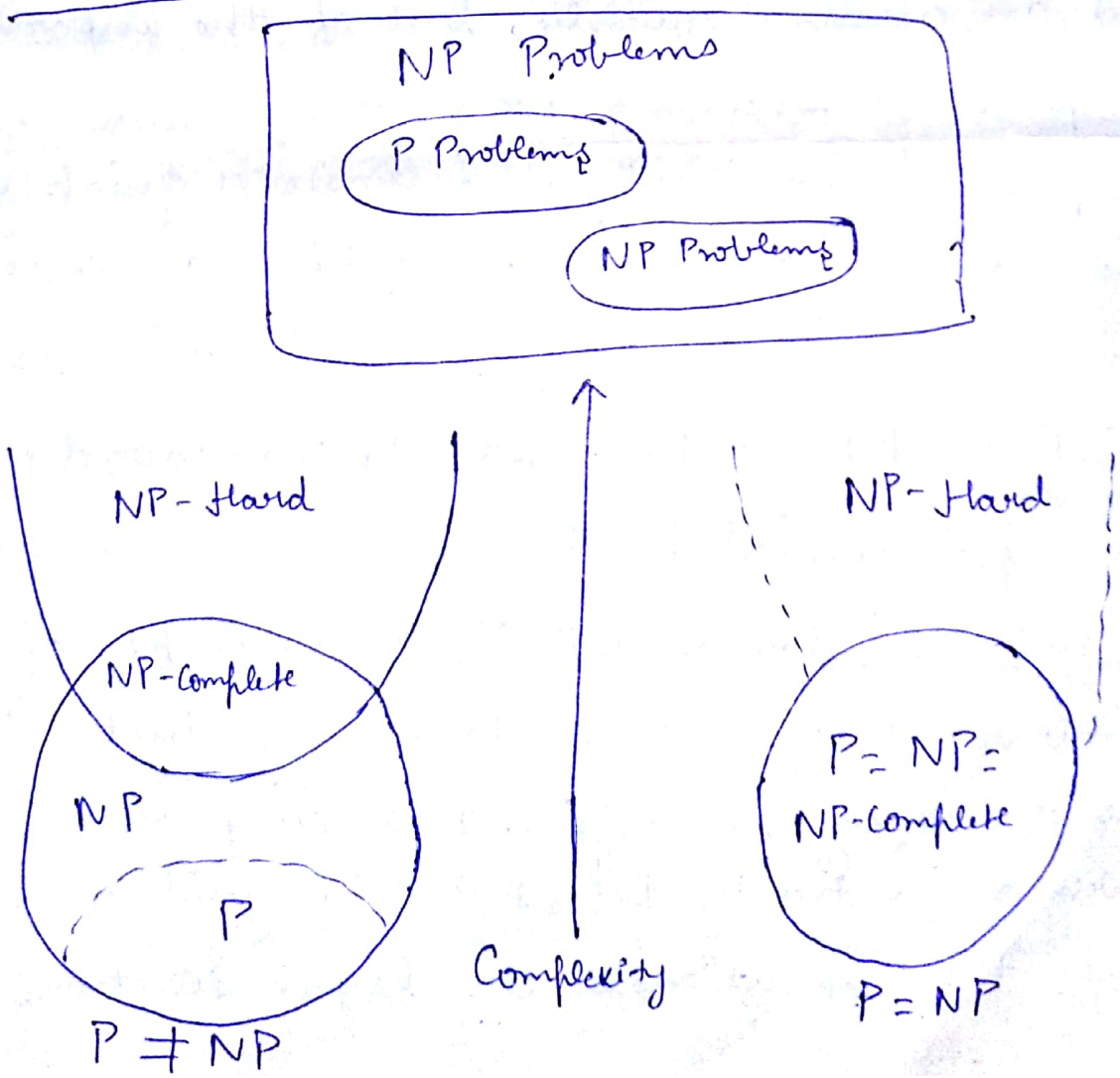
$$(S, \bar{S}) = \{(v, t)\}$$

and the capacity of the cut is $c(v, t) = 2$. The green flow, of value 2, is a maximum flow. Note that it is different to the flow in Example 1, even though it has the same value.

* Computational Complexity: It is a branch of the theory of computation in computer science that focuses on classifying computational problems according to their inherent difficulty and relating those classes to each other.

A computational problem is understood to be a task that is in principle amenable to being solved by a computer i.e. the problem may be solved by mechanical application of mathematical steps such as an algorithm.

* POLYNOMIAL V/S NON-POLYNOMIAL COMPLEXITY:

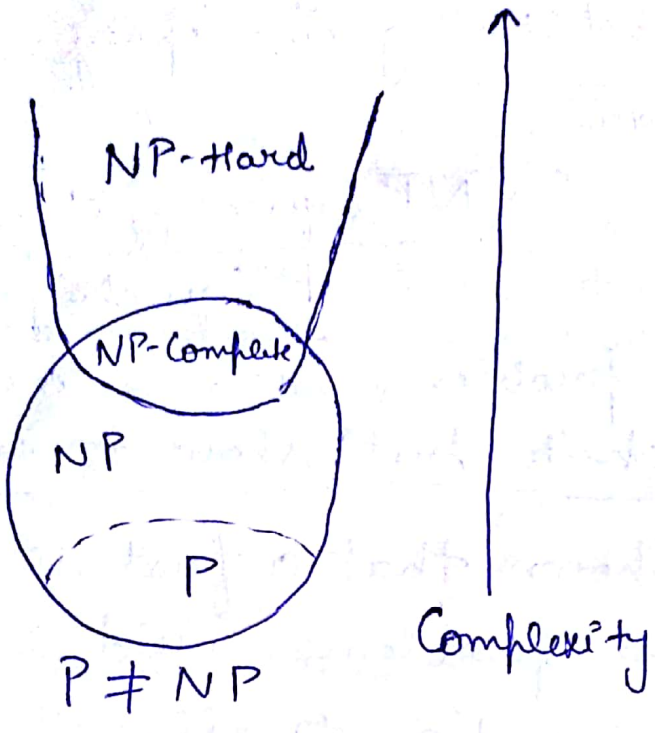


UNSOLVED PROBLEM IN COMPUTER SCIENCE:

- If the solution to a problem is easy to check for correctness, is the problem easy to solve?
- The P versus NP problem is a major unsolved problem in computer science.
- It asks whether every problem whose solution can be quickly verified i.e. in polynomial time can also be solved quickly i.e. in polynomial time.
- The general class of algorithms which can provide an answer in polynomial time is called "CLASS P".
- For some problems, there is no known way to find an answer quickly, but if the information is provided showing what the answer is, it is possible to verify the answer quickly. The class of questions for which an answer can be verified in polynomial time is called "CLASS NP" which stands for nondeterministic polynomial time.

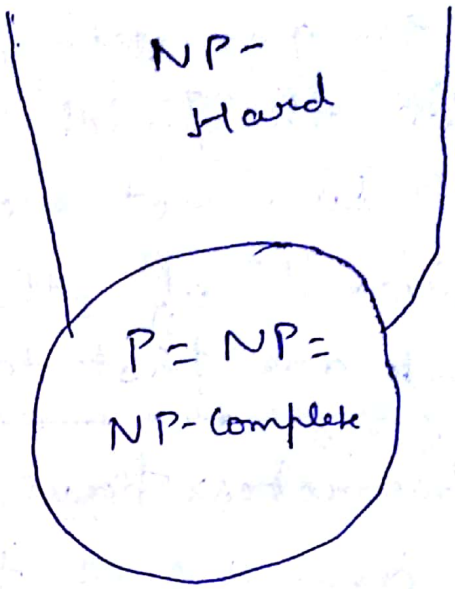
For eg: Consider SUDOKU, an example of a problem that is easy to verify, but whose answer may be difficult to compute. Given a partially filled Sudoku grid of any size, there is atleast one legal solution.

* NP HARD CLASSES:



$P \neq NP$

It is valid under the assumption that $P \neq NP$



$P = NP$

It is valid under the assumption that $P = NP$

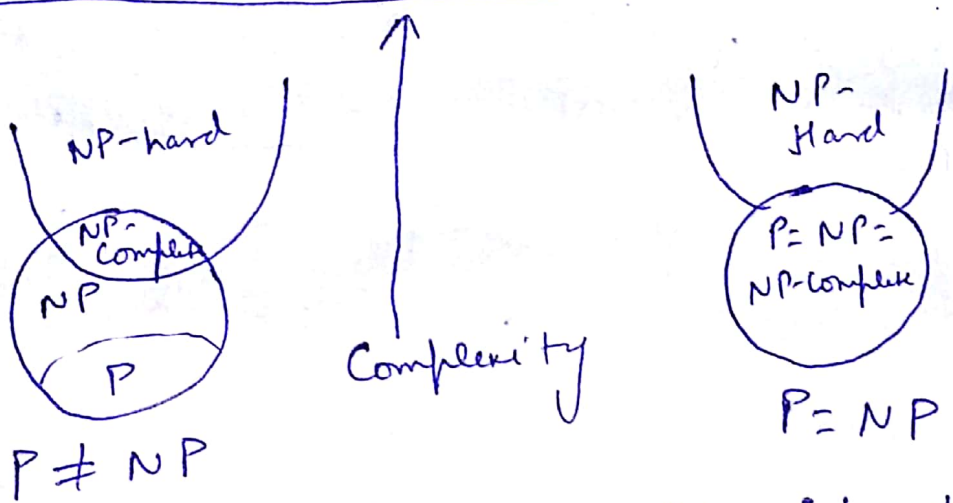
- NP-hardness i.e. non-deterministic polynomial time hardness in computational complexity theory is the defining property of a class of problems that are atleast as hard as the hardest problem in NP.
- The problems that are harder to compute than to verify usually comes under this category. Eg- SUDOKU, Circuit Satisfiability, etc.
- A problem 'H' is NP-hard, when every problem 'L' in NP can be reduced in polynomial time to 'H'. i.e. assuming a

- A proposed solution can be easily verified and the time to check a solution grows polynomially as the grid gets bigger.
- So, the Sudoku is in NP (quickly checkable) but does not seem to be in P (quickly solvable). Thousands of other problems seem similar, that are fast to check but slow to solve.
- Researches have shown that a fast solution to any one of these problems could be used to build a quick solution to all others. These problems possess a property known as NP - Completeness.
- Also, P = NP determine whether problems that can be verified in polynomial time like Sudoku can also be solved in polynomial time.
- If it turned out that P ≠ NP, it means that there are problems in NP that are harder to compute than to verify i.e. they could not be solved in polynomial time but the answer could be verified in polynomial time. It comes under the category of NP-hard problems.

solution for 'H' takes 1 unit time, we can use. H's solution to solve 'L' in polynomial time.

- Consequently, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP.
- Also, the class 'P' in which all the problems can be solved in polynomial time are contained in the NP-class.

* NP-COMPLETE CLASSES:



- In the computational complexity theory, an NP-complete decision problem is the one, belonging to both the NP and the NP-hard complexity classes.
- The set of non-deterministic polynomial time complete problems is denoted by NP-C or NPC.

- Although, any given solution to an NP-complete problem can be verified quickly in polynomial time, there is no known efficient way to locate a solution in the first place.
- The time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows.
- While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, the NP-complete problems are frequently encountered which are often addressed using heuristic methods & approximation algorithm
- Various examples of NP-complete problems can be stated as:
  - The Clique problem
  - The Vertex Cover problem
  - The Hamiltonian Cycle problem
  - The Travelling Salesman problem
  - The Subset Sum problem