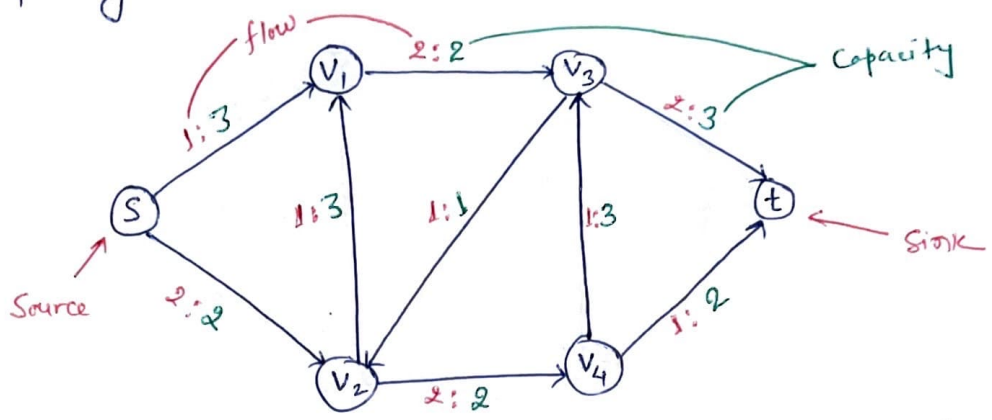
Flow Network Algorithm Design - 2.

Def:- A Graph, $G=(V,E)$, where each edge has a non-negative capacity 'C', and there is a source vertex, s and sink vertex t.

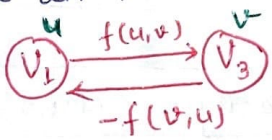


Flows in 'G' satifies these 3 properties:-

i) Capacity Constraint: For all $u,v \in V$, we require $f(u,v) \leq c(u,v)$

ii) Skew Symmetry: for all $u,v \in V$, we require $f(u,v) = -f(v,u)$

Meaning:- If there is a ~~flow~~ path from u to v having flow $f(u,v)$ then there will be another path in opposite direction with negative flow $-f(v,u)$.



iii) Flow Conservation:- For all $u \in V - \{s,t\}$, we require $\sum_{v \in V} f(u,v) = 0$

Meaning:- Any particular vertex the flow will not accumulated. i,e; whatever is going into the vertex, that will be comming out from that vertex.

Ex:- At V_1 there are 2 flow incoming and 2 flow is outgoing.

Note:- There is a cycle in the graph. i/e.

$V_2 \rightarrow V_4 \rightarrow V_3$

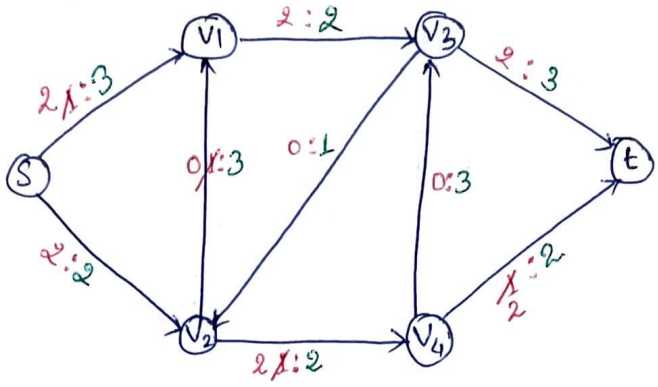
If you subtract an equal amount from each and every edge of the cycle, there will not be any difference. It will still follow all of the above three ~~two~~ properties.

NOTE:- The quantity $f(u,v)$, which can be positive, zero or negative is called the flow from vertex u to v. The value of a flow f is defined as

This denote the flow value $\leftarrow$ $|f| = \sum_{v \in V} f(s,v)$, that is, the total flow out of the source.

Max flow problem:-

Aim:- We want maximum flow from source to sink, without violating the three properties of flow.



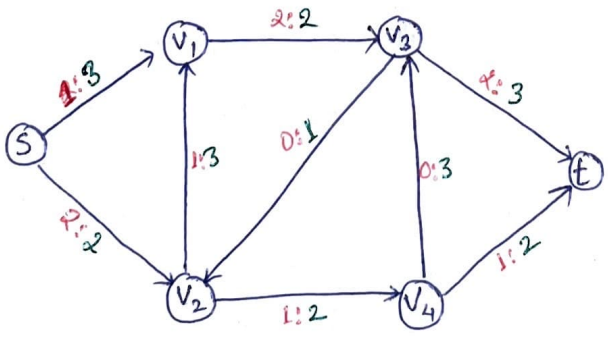
Current flow = 2+1 = 3

Is there any edge where we can decrease the flow to increase the overall flow of the network?

New current flow = 2+2 = 4

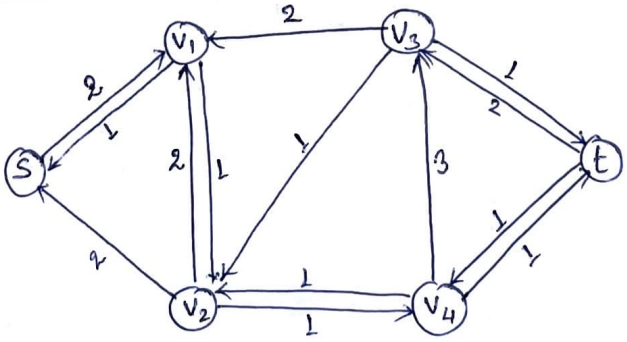
If there are 10,000 edges in a flow network then it will be very hard to identify which edge's flow to reduce to increase the overall flow. So, here comes the residual graph concept into the picture.

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



Flow = 2+1 = 3

Residual Graph:



If there is a path from source S to sink t in the residual graph, then there is a scope to increase the overall flow of the flow network.

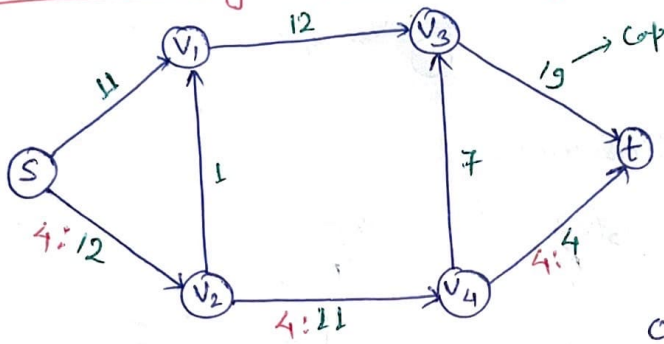
Possible path.

$S \xrightarrow{2} V_1 \xrightarrow{1} V_2 \xrightarrow{1} V_4 \xrightarrow{1} t$ ← Augmenting path

$\min(2,1,1,1) = 1$ → Bottleneck capacity.

Now flow = 3+1 = 4

Max Flow Problem Using Ford Fulkerson Algorithm



→ Capacity

← Flow network

← Residual Graph in initial phase. b/s the flow is zero initially

Overall flow = 0

Algorithm:

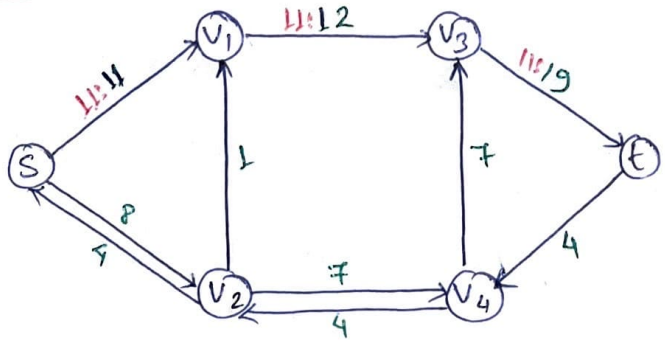
Ford-FULKERSON-METHOD(G, S, t)

1. Initialize flow f to 0.
2. while there exists an augumenting path p
3. do augument flow f along p
4. return f.

Augumenting path 1: $S \xrightarrow{12} V_2 \xrightarrow{11} V_4 \xrightarrow{} t$ (4) → min

Overall flow = 0+4 = 4

Next Residual Graph

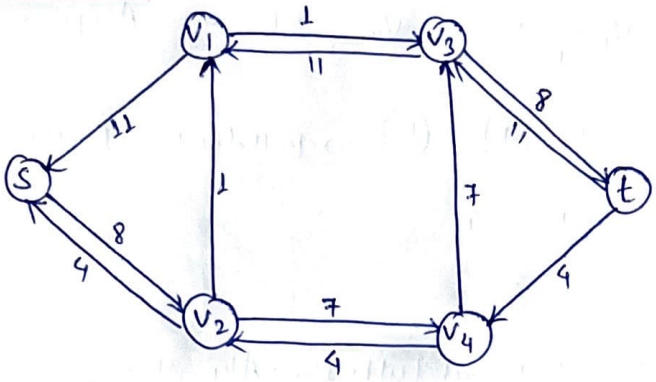


Augumenting path 2:- $S \xrightarrow{11} V_1 \xrightarrow{12} V_3 \xrightarrow{19} t$

min.

Overall flow = 4+11 = 15

Next Residual Graph:

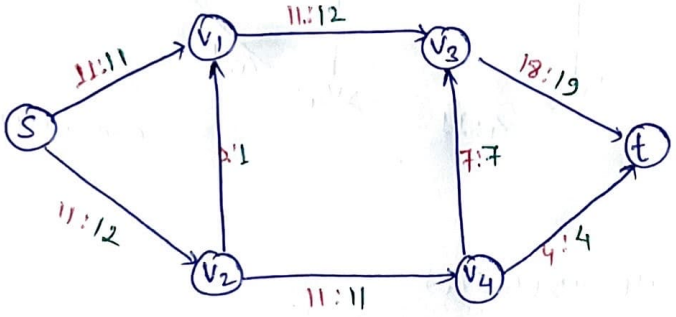


Augumenting path 3: $S \xrightarrow[8]{} V_2 \xrightarrow[7]{} V_4 \xrightarrow[8]{} V_3 \longrightarrow t$

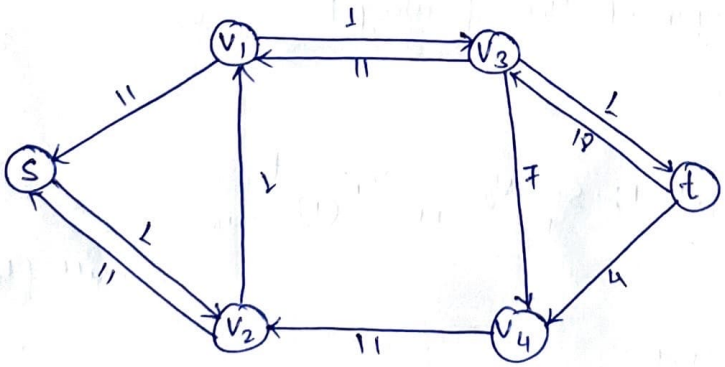
7 → min

So, the network flow will be.

Overall flow = 15 + 7 = 22



Next Residual Graph:

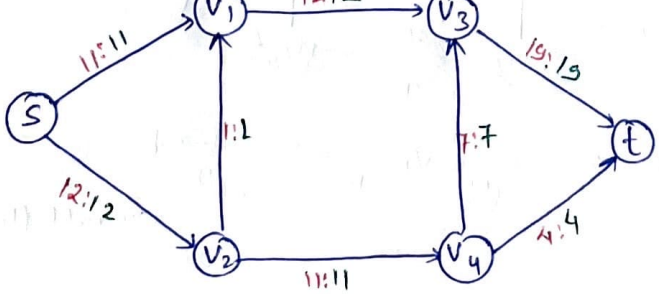


Augumenting path 4: $S \xrightarrow[1]{} V_2 \longrightarrow V_1 \xrightarrow[1]{} V_3 \longrightarrow t$

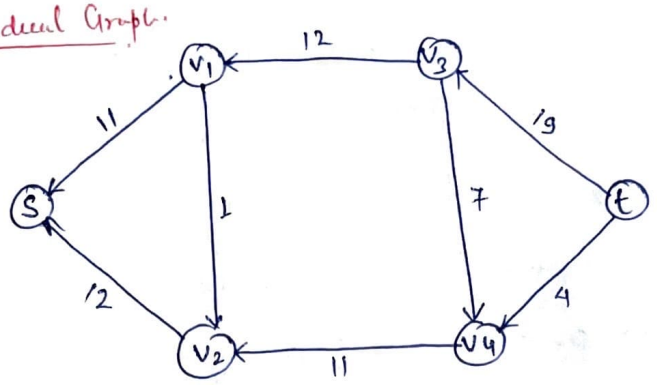
1 → min.

Overall flow = 22 + 1 = 23

New network flow will be.



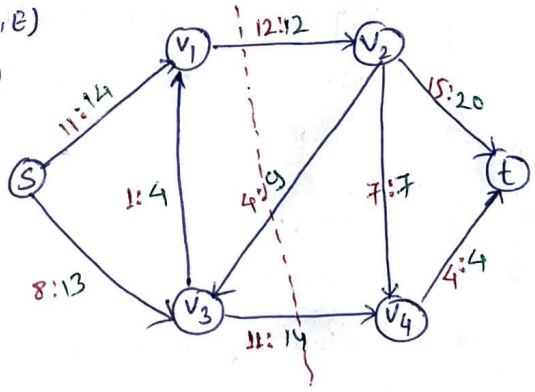
Next Residual Graph.



As this residual graph doesn't have any augumenting path from source s to sink t. There are only incomming edges to the source, so there will be no further flow in this residual graph. Hence, the finall overall flow of the given network graph ~~flow~~ is 23. This is the maximum flow of the network.

Max-flow Min Cut Theorem. (S-T) cut.

Def: 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 the ~~cut~~ flow, then the net flow across the cut (S,T) is denoted by f(S,T) and it can be calculated as

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

$$= 12 + 11 - 4$$

$$= 19$$

$S = \{s, v_1, v_3\}$

$T = \{t, v_2, v_4\}$

→ The capacity of the cut (S,T) is denoted a c(S,T)

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

$$= 12 + 14$$

$$= 26$$

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 max flow in G.

2) The residual network G_f contains no augumenting paths.

3) $|f| = c(S,T)$ for some cut (S,T) of G.

We will use the rule of transitive to prove the above statements to prove the ford-fulkerson algorithm.

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

Proof:

$1 \Rightarrow 2$

Suppose for the sake of contradiction that f is the max flow in G but G_f has an augumenting path P. Then we can augement $c_f(P) > 0$ unit along P and get a better flow, which contradicts that f is the max flow.

Proof

$2 \Rightarrow 3$

We need to prove that $|f| = c(S,T)$.

Suppose G_f has no augumenting path i.e. G_f contains no path from s to t.

Define $S = \{ v \in V :$ there exists a path from s to v in $G_f \}$

$S = \{s, a, c\}$

$T = \{t, b, d, e\} = (V - S)$

We will have a cut (S,T)

→ There will be edges denoted by (u,v) where $u \in S$ & $v \in T$

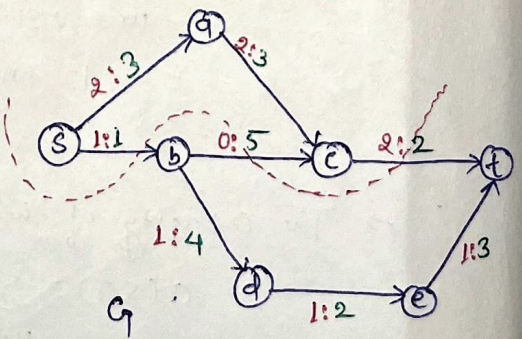
$f(u,v) = c(u,v)$ {Eg. $S \to b$, $c \to t$}

otherwise v will be placed in s.

→ There will be some edges denoted by (u,v) where $u \in T$ and $v \in S$

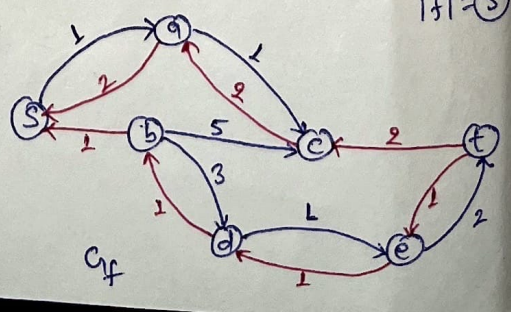
$f(u,v) = 0$

otherwise $c_f(u,v) > 0$ and v should be in S



G

$|f| = 3$



G_f

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

$$= \sum_{u \in S} \sum_{v \in T} f(u,v) - 0$$

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

$$\boxed{|f| = c(S,T)}$$

Proof: 3 $\Rightarrow$ 1.

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

i.e $|f| \leq c(S,T)$ for all cuts (S,T).

Let (S,T) be a cut in G and f be any flow.

$$|f| = f(S,T)$$

← Here we have to prove that f is the max flow.

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

← Conside as ommited.

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

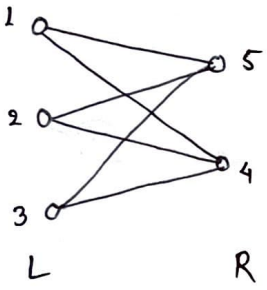
$$= c(S,T)$$

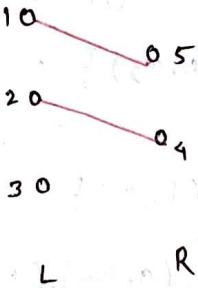
The condition $|f| = c(S,T)$ this implies that f is the max flow.

Maximum Bipartite Matching

Def: Given an undirected graph $G=(V,E)$, a matching is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is matched by matching M if some edges in M is incident on v; otherwise, v is unmatched.

→ A maximum matching is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$.





1, 2, 4, 5 → Matched vertex

3 → Unmatched vertex.

$|M| = 2$

↓

Maximum matching in bipartite graph.

Augmenting Path: Augmenting path is an alternating path such that the path starts and ends with unmatched vertex.

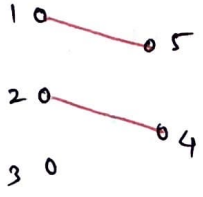
Augmenting Path Algorithm:

Input: A graph $G = (L \cup R \in E)$

output: Maximum matching M.

1. $M = \emptyset$
2. While there exists an augmenting path P w.r.t M
3. $M = M \Delta P$ → Symmetric Difference.
4. return M.

$$M \Delta P = (M - P) \cup (P - M)$$



output ↓

$M = \{(1,5), (2,4)\}$

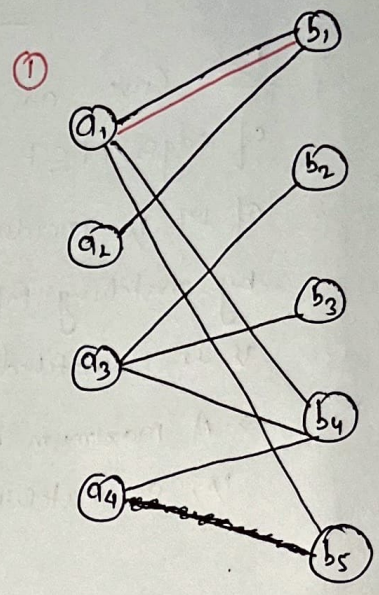
Illustration:-

① → $M = \phi$ $\quad |M| = 0$

$P = (a_1, b_1)$

$M = M \Delta P = (M-P) \cup (P-M)$

$= \{(a_1, b_1)\}$; $|M| = 1$.



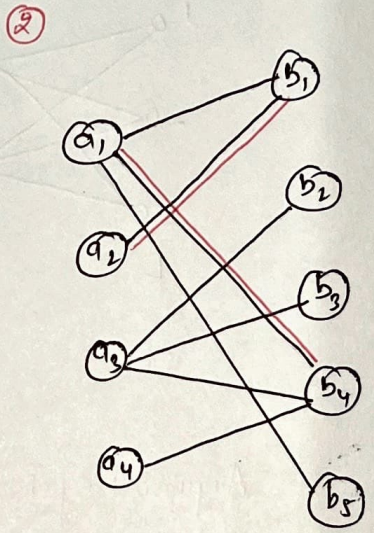
② → $M = \{(a_1, b_1)\}$

$P = \{(b_4, a_1), (a_1, b_1), (b_1, a_2)\}$

(4m) (4m)

$M = M \Delta P = (M-P) \cup (P-M)$

$= \{(b_4, a_1) (b_1, a_2)\}$; $|M| = 2$



③ → $M = \{(b_4, a_1) (b_1, a_2)\}$

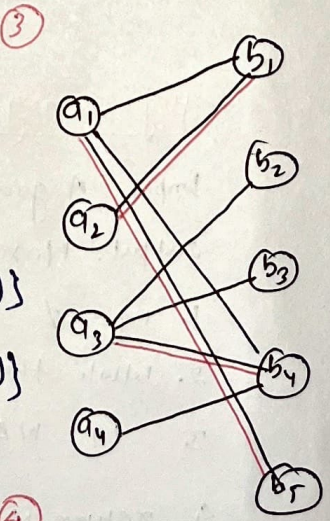
$P = \{(b_5, a_1), (a_1, b_4), (b_4, a_3)\}$

(4m) (M) (4m)

$M = M \Delta P = (M-P) \cup (P-M)$

$= \{(b_1, a_2)\} \cup \{(a_1, b_5), (b_4, a_3)\}$

$= \{(b_1, a_2), (a_1, b_5), (b_4, a_3)\}$; $|M| = 3$



④ $M = \{(b_1, a_2), (a_1, b_5), (b_4, a_3)\}$

$P = \{(b_2, a_3), (a_3, b_4), (b_4, a_4)\}$

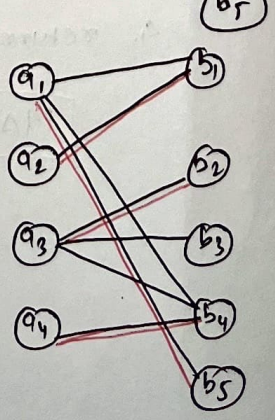
(4m) (M) (4m)

$M = M \Delta P = (M-P) \cup (P-M)$

$= \{(b_1, a_2), (a_1, b_5)\} \cup \{(b_2, a_3), (b_4, a_4)\}$

$= \{(b_1, a_2), (a_1, b_5), (b_2, a_3), (b_4, a_4)\}$

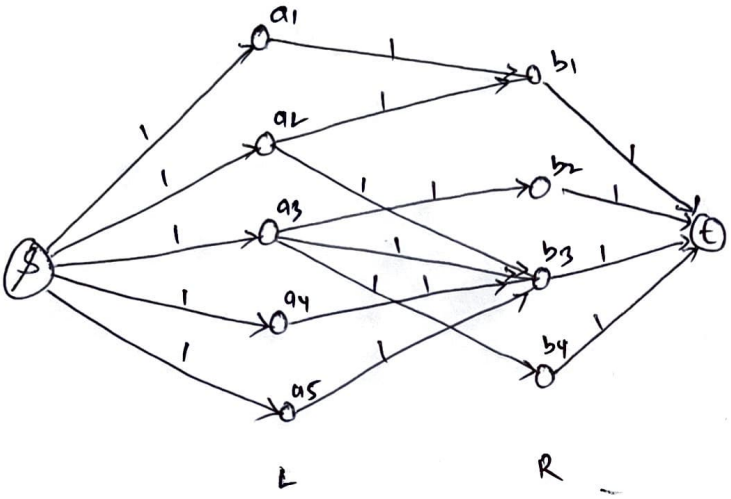
$|M| = 4$



No augmenting Path. further.

STOP!

Maximum Matching Using Ford-Fulkerson Algorithm.



Bipartite Graph $G = (V, E)$

$= (L \cup R, E)$

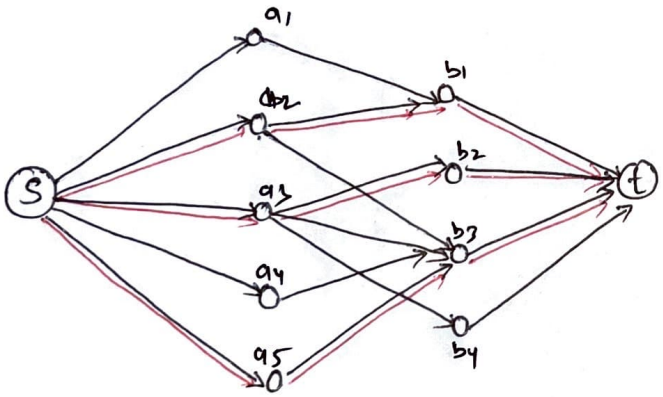
↓ Converted to a flow network

$G' = (V', E')$

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

↓ flow network

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



G' as directed graph and assign 1 as the capacity on all edges.

↓

Execute Ford-Fulkerson Algo.

$M = \{(a_2, b_1)(a_3, b_2)(a_5, b_3)\} = 3$

$|f| = 3$

Reductions for Algorithms

Reduction are an honorably way to generate new algorithm from the old ones. Whenever we can translate the input for a problem we want to solve into input for a problem we know how to solve, we can compose the translation and the solution into an algorithm to deal with our problem.

→ Reduction will lead to the efficient algorithm.

→ To solve problem A

↓

Reduce the A instance to an instance of B,

↓

Solve this instance using an efficient algo. for problem B.

* Overall running time = Time for reduction + Time for solve instance B.

1. Closest Pair:-

The closest pair problem asks to find the pair of numbers within a set S that have the smallest difference b/w them.

Ex: $S = \{10, 4, 8, 3, 12\}$

Closest Pair is $(3, 4)$.

Now, we can make it a decision problem by asking if this value is less than some threshold:

Input:- A set S of n numbers, and threshold t.

Output:- Is there a pair $s_i, s_j \in S$ such that $|s_i - s_j| \leq t$?

Soln Finding the closest pair is a simple application of sorting, since the closest pair must be neighbouring elements after sorting.

Algo:-
```
CloseEnoughPair(S, t)
Sort S
Is min_{1≤i<n} |s_{i+1} − s_i| ≤ t ?
```

Observations:

1) Decision Version Vs General Problem:- The decision version of the closest-pair problem focuses on whether there exist a pair of numbers in the set with a difference less than a given threshold. This version captures the essence of what's interesting about finding the closest pair, but it is not easier than finding the actual closest pair.

2) Sorting Complexity:- Sorting the set is crucial; a fast sort followed by a linear scan yields $O(n \log n + n)$ complexity.

3) Lower bound on Sorting:- While sorting generally takes $\Omega(n \log n)$ time, this directly doesn't prove the same for finding a close-enough pair.

4) Sorting algorithm Implications:- If finding a closest pair took at least $\Omega(n \log n)$ time, it would imply the same for sorting, as faster sort would contradict the problem's complexity.

2. Longest Increasing Subsequence:-

Input: An integer or character sequence S.

Output: What is the length of longest sequence of positions $P_1, \ldots P_m$ such that $p_i < p_{i+1}$ and $S_{p_i} < S_{p_{i+1}}$?

Algo:
```
LongestIncreasingSubsequence(S)
    T = Sort(S)
    C_ins = C_del = 1.
    C_sub = ∞
    Return(|S| - EditDistance(S, T, C_ins, C_del, C_sub)/2)
```

Observations:

1. Reduction to edit distance.
   - → The LIS problem can be solved using the edit distance problem.
   - → The sequence T is constructed by sorting the elements of S in increasing order. This ensures that any common subsequence b/w S and T must be an increasing subsequence.
   - → By setting $C_{sub} = \infty$, substitution are disallowed, meaning only insertion and deletion are allowed in the alignment.

2. Optimal Alignment :-

→ The optimal alignment of S & T finds the longest common subsequence and remove everything else.

→ For example, transforming S = cab to T = abc costs two operations: inserting a and removing c.

→ Subtracting half of the cost from the length of S gives the length of LIS.

3. Implication of Reduction :-

→ This reduction has implication on the complexity of finding LIS.

→ Since sorting takes O(nlogn) time, the overall complexity is quadratic due to the O(|S|·|T|) time complexity of edit distance.

→ While this provides a simple polynomial-time algorithm, it is not the most efficient. There exists a faster O(nlogn) algorithm for LIS using advanced data structures.

→ Edit distance, on the otherhand, is known to be quadratic in the worst case.

In essence, this reduction demonstrates a way to solve LIS problem using edit distance, but it may not be the most optimal approach. While it provides a polynomial-time solution, there are faster algorithms available specifically designed for LIS.

3. Least Common Multiple :-

The least common multiple (LCM) and greatest common divisor (GCD) problem arise ~~from~~ often in working with integers. We say b divides a (written b|a) if there exists an integer d such that a = bd. Then;

Problem: Least Common Multiple (LCM)

Input: Two positive integer x and y.

output: Return the smallest positive integer m such that m is a multiple of x and also a multiple of y.

Problem: Greatest Common Divisor (GCD)

Input :- Two positive integer x and y.

output :- Return the largest integer d such that d divides by x and y.

For example:-

LCM(24,36) = 72 and GCD(24,36) = 12,

Both problem can be solved easily after reducing x and y to their prime factorization, but no efficient algorithm is known for factoring integers.

1. Euclid's Algorithm for GCD :-

→ Euclid's algorithm is a recursive method that finds the greatest common divisor of two numbers.

→ It relies on the fact that if b divides a (denoted as b|a), then gcd(a,b) = b.

→ It repeatedly replaces (a,b) by (b, a mod b) until b = 0. This process efficiently computes the GCD, with worst-case time complexity of O(log b).

2. Observation for LCM :-

→ The LCM of two numbers x and y is defined as the smallest positive integer that is a multiple of both x and y.

→ Since $x \cdot y$ is a multiple of both x and y, we know that LCM(x,y) is less than or equal to $x \cdot y$.

3. Using Euclid's Algorithm for LCM :-

→ If we divide $x \cdot y$ by their gcd, it essentially removes any non-trivial factors shared between x and y.

→ So, we compute $LCM(x,y) = \frac{x \cdot y}{GCD(x,y)}$

→ This is because any common factors shared between x and y are removed by dividing the GCD, leaving the smallest common multiple.

This reduction gives us a nice way to reuse Euclid's efforts for LCM.

4. Convex Hull :-

→ A polygon is convex if the straight line segment drawn b/w any two points inside the polygon P lies completely within the polygon.

→ The convex hull provides a very useful way to provide structure to a point set.

Problem: Convex Hull.

Input : A set S of n points in the plane.

output : Find the smallest convex polygon containing all the points of S.

Soln.

i) Transformation of Input :-

→ Each number in the original set is mapped to a point in the plane. Specifically, the number x is mapped to the point (x, x^2)

→ This mapping results in each integer being represented by a point on the parabola $y = x^2$.

ii) Convexity of the Region :-

→ The region above the parabola $y = x^2$ is convex. This property ensures that every point in the mapped set lies on the convex hull

→ Additionally, since neighbouring points on the convex hull have neighbouring x values, the convex hull returns the points sorted by their x-coordinates, which corresponds to the original numbers

iii) Algorithm :-

→ Given the mapped points, the convex hull algorithm applied to find the convex hull of this point set.

→ The points on the convex hull are then read off from left to right, representing the sorted numbers in the original set.

iv) Implication :-

→ The reduction implies a connection b/w the time complexity of sorting and computing the convex hull.

→ If we could compute the convex hull in better than $O(n \log n)$ time this reduction would imply a sorting algorithm faster than $O(n \log n)$ which contradicts the known lower bound for sorting.

→ Therefore, the convex hull problem must also have a time complexity of $\Omega(n \log n)$.

→ Additionally, any $O(n \log n)$ convex hull algorithm, when coupled with the reduction, provides a complicated but correct $O(n \log n)$ sorting algorithm.

This reduction demonstrates a relationship between the computational complexity of sorting and computing the convex hull, suggesting that they both require $\Omega(n \log n)$ time.

Hamiltonian cycle to TSP Reduction:-

Problem: Traveling Salesman Problem.

Input: Pairwise distance b/w n cities and a budget b.

Output: A cycle that visits each vertex exactly once and has total length atmost b.

Def:- TSP is a search problem: given a sequence of vertices, it is easy to check whether it is a cycle visiting all the vertices of total length at most b.

→ TSP is usually stated as an optimization problem: we stated its decision version to guarantee that a candidate solution can be efficiently checked for correctness.

PerTSP!-

→ Check all permutation: $O(n!)$
Extremely slow.

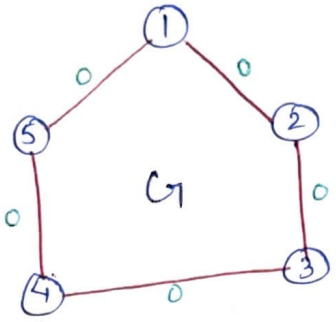
→ Dynamic Programming: $O(n^2 2^n)$

→ No significantly better upper bound is known.

→ No. of TSP solutions: $\frac{(n-1)!}{2}$

Problem type Difficulty	verifiable in P time	Solvable in P time
i) P	Yes	Yes
ii) NP	yes	Yes / No
iii) NP Complete	Yes	Unknown
iv) NP-Hard	Yes / No	Unknown.

Reduction of Hamiltonian Circuit (G) to TSP(G) in polynomial time

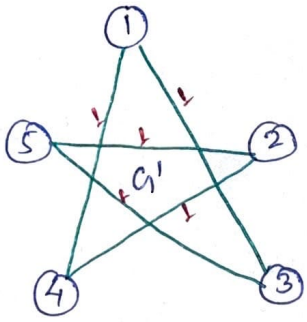


Hamiltonian Circuit (G)

$G(V, E)$, Hamiltonian Path = $1 \to 2 \to 3 \to 4 \to 5$

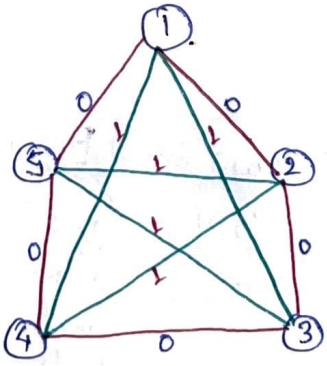
To form a TSP:-

Step 1: Construct a complement graph G' (takes O(N) time)



$1 \to 4 \to 2 \to 5 \to 3$

Step 2: Construct a complete graph by combining G and G'



Define cost function: $C(i,j) = \begin{cases} 0; & \text{if } (i,j) \in G, \\ 1; & \text{if } (i,j) \in G' \end{cases}$

NOTE: Converting unweighted G to weighted G' takes linear time O(N)

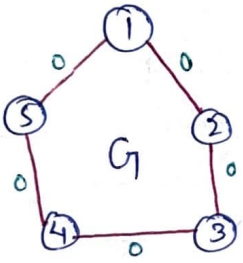
Hamiltonian cycle problem reduced to an instance of TSP:-

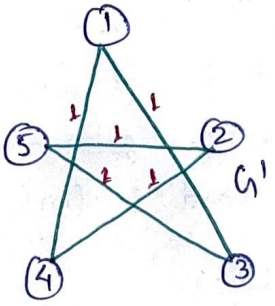
→ The actual reduction is quite simple, with the translation from unweighted to weighted graph easily performed in linear time.

→ Further, this translation is designed to ensure that the answer of the two problem will be identical.

→ If the graph G has a hamiltonian cycle, then this exact same tour will correspond to n edges in E', each with weight 1. Therefore, this gives a TSP tour of G' of weight exactly n.

→ If G doesn't have a hamiltonian cycle, then there can be no such TSP tour in G', because the only way to get a tour of cost n in G' would be to use only edges of weight 1, which implies a Hamiltonian cycle in G.





TSP tour : $1 \to 4 \to 2 \to 5 \to 3 \to 1$

weight = $n = 5$

Observations:-

→ While NP is the verification if a solution works. It's a decision problem, answered by yes or no in polynomial time.

→ Example:- we can verify that $1 \to 2 \to 3 \to 4 \to 5 \to 1$ is a solution to know if a graph is a hamiltonian cycle.

→ That is, we can easily verify that 1, 2, 3, 4, 5 are vertices of the graph and there exists edges that make a cycle. But that doesn't mean that there exists a solution that finds the Hamiltonian cycle in polynomial time (especially when n approaches infinity)

→ It belongs to NP-hard problem, that makes it NP-Complete.

→ Finally, by reducing TSP to hamiltonian circuit problem, TSP also becomes an NP-complete problem; ie we can easily verify that 1, 2, 3, 4, 5 are vertices of the graph and there exists edges that make a cycle in least cost. But that doesn't mean that there exists a solution that find the TSP in P time.

Independent set and Vertex Cover.

Independent set Problem:-

For a graph $G=(V,E)$, a set of nodes $S \subseteq V$ is called independent set if no two nodes in S are connected by an edge $e \in E$. The independent set problem is to find the largest independent set in a graph. It is not hard to find small independent set, e.g. a trivial independent set is any single node, but it is hard to find the largest independent set.

Problem: Independent set

Input: A graph G and integer $k \leq |V|$.

Output: Does there exist a set of k independent vertices in G?

Vertex Cover Problem:-

Given a graph $G=(V,E)$, a set of nodes $S \subseteq V$ is called vertex cover if every edge $e \in E$ has at least one end in S. It is not hard to find large vertex covers, e.g. a trivial vertex cover is the set $S=V$. However, it is hard to find small vertex cover.

Problem:- Vertex Cover

Input:- A graph $G=(V,E)$ and integer $k \leq |V|$.

Output: Is there a subset s of atmost k vertices such that every $e \in E$ containg at least one vertex in S?

Proof:-

→ If S is an independent set for a given graph $G=(V,E)$, then for any edge $e=(u,v)$ where $e \in E$, at most one of the vertices u and v is in S. Hence, at least one of them must be in V-S. Thus, every edge has at least one end in V-S. So, V-S must be a vertex cover.

Independent set

$S=\{B,C,E,G\} \leftarrow IS$

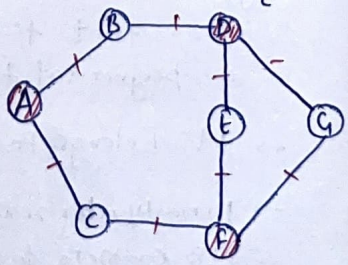
$V-S=\{A,D,F\}$

↑ vertex cover

~~Vertex Cover =~~

~~$S=\{A,D,F\}$~~

~~$S=V-\{B,C,E,G\}$~~



→ Conversely, if V-S is a vertex cover, if any two nodes u, and v in S were connected by an edge e, then ~~neighbor~~ neither u nor v would be in V-S, which contradicts the initial assumption that V-S is a vertex cover. That is, no two nodes in S can be adjacent and hence S is an independent set.

→ Given the fact presented above, (S being independent set iff the set V-S is a vertex cover.), we can conclude that the two problem is polynomially reducible to the other.

* Independent set $\leq_p$ Vertex Cover.

Suppose that we have an efficient algorithm for solving Vertex cover, it can simply be used to decide whether G has an independent set of size at least k by asking it to determine whether G has a vertex cover of size at most n-k.

* Vertex Cover $\leq_p$ Independent set.

Suppose that we have an efficient algorithm for solving Independent set, it can simply be used to decide whether G has a vertex cover of size at most k, by asking it to determine whether G has an independent set of size at least n-k.

Satisfiability Problem

* **Boolean formulas:**

→ Variables: $x_1, x_2, x_3 \dots x_n$ (can be either true or false)

Literals → Terms: $t_1, t_2, t_3 \dots t_n$ (t_i is either x_i or $\bar{x}_i$)

→ clauses:- $t_1 \vee t_2 \vee \dots t_n$ (A clause is true if any term in it is true)

Ex:- $(x_1 \vee \bar{x}_2)\ (\bar{x}_1 \vee \bar{x}_3)\ (x_2 \vee \bar{x}_3)$

literals

clauses

Def: A truth assignment is a choice of true or false for each variable i.e, a function $y: x \rightarrow \{\text{true}, \text{false}\}$.

Conjunctive Normal form (CNF):-

A CNF formula(ϕ) is a conjunction of ~~clauses~~ disjunction (clauses) of literals, where a literal is a variable or its compliments.

$C_1 \wedge C_2 \wedge C_3 \dots \wedge C_n$

$\phi(x_1, x_2 \dots x_3) =$

Eg:- $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_3)$

Satisfiability Problem (SAT):-

Find an assignment to the variables $x_1 \dots x_n$ such that $\phi(x_1 \dots x_n) = 1$ or prove that no such assignment exists.

OR.

Given a set of clauses $C_1 \dots C_k$ over variable $X = \{x_1 \dots x_n\}$ is there a satisfying assignment?

Ex:- $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_2 \vee \bar{x}_3)$.

→ When we assign all variable to 1, then this CNF is not satisfiable.

→ When we assign all variable to 0, this CNF is satisfiable.

3-SAT Problem:-

Given a set of clauses $C_1, \dots C_k$, each of length 3, over a set of variable $X = \{x_1, \dots x_n\}$, does there exist a satisfying truth assignment?

Reduction of 3-SAT Problem:-

Theorem: 3-SAT $\leq_p$ Independent set

Proof: Suppose we have an algorithm to solve Independent set, how we can use it to solve 3-SAT problem?

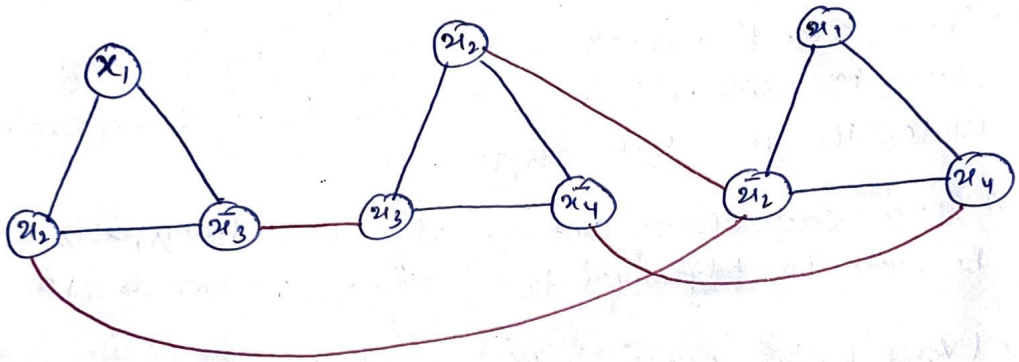
To solve 3-SAT:-

→ you have to choose a term from each clause to set to true.

→ But you cannot set both x_i and $\bar{x}_i$ to true.

For ex:

$$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_2 \vee x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2 \vee x_4)$$



Theorem: This graph has an independent set of size k iff the formula is satisfiable.

Proof:-

→ If the formula is satisfiable, there exist at least one true literal in each clause. Let S be a set of one such true literals from each clause $|S| = k$ and no two nodes in S are connected by an edge.

→ If the graph has an independent set S of size k, we know that it has one node from each "clause triangle". Set these terms to true. This is possible because no 2 are negations of each other.

3-SAT is NP-Complete:-

To prove 3-SAT is NP-Complete, we need to demonstrate two things:

i) 3SAT is NP

ii) Every NP problem can be reduced to 3SAT in polynomial time.

Proof

i) 3SAT is NP:-

→ To show that 3SAT is in NP, we need to demonstrate that given a proposed solution (a truth assignment), we can verify in polynomial time whether it satisfies the given 3SAT instances.

→ Given a truth assignment, we can simply go through each clause in 3SAT instance and check if the clause evaluates to true under the given truth assignment.

→ Since each clause contains at most 3 literals, this verification can be done in polynomial time. Therefore, 3SAT is NP.

ii) Every problem in NP can be reduced to 3SAT in polynomial time:-

→ This is usually shown by reducing a known NP-complete (such as the Boolean satisfiability problem) to the problem in question (3SAT in this case) in polynomial time.

→ Since SAT is NP-complete, we can reduce SAT to 3SAT in polynomial time.

→ The reduction involves transforming each clause in the SAT instance into clauses of at most 3 literals. This reduction can be done in polynomial time because it only requires duplicating clauses and introducing additional variables where necessary to break larger clauses in to smaller ones.

Ex: Suppose we have the following SAT instance:

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_3 \vee x_4 \vee x_5)$$

To reduce this SAT instance to an equivalent 3SAT instance, we will introduce a new variable and clauses as needed:-

i) The first and third clauses already have at most 3 literals, so we leave them unchanged.

ii) For the second clause $(\bar{x}_1 \vee x_2)$, since it has only 2 literals, we introduce a new variable z_1 and rewrite it as $(\bar{x}_1 \vee x_2 \vee z_1) \wedge (\bar{z}_1 \vee z_1)$ to make it 3SAT clause.

Now our modified 3SAT instance becomes:-

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee z_1) \wedge (\bar{z}_1 \vee z_1) \wedge (\bar{x}_3 \vee x_4 \vee x_5)$$

In this modification, each clause has at most 3 literals, and it is equivalent to the original SAT instance. This transformation can be done in polynomial time.

Therefore, we have shown an example of reducing SAT to 3SAT, demonstrating that 3SAT is NP-complete.

NOTE:-

$(\bar{x}_1 \vee x_2 \vee z_1) \wedge \overbrace{(\bar{z}_1 \vee z_1)}^{\text{Tautology}}$ is equivalent to original clause $(\bar{x}_1 \vee x_2)$ because both expression are satisfied when $(\bar{x}_1 \vee x_2 \vee z_1)$ is true, and the presence of tautology in the combined expression does not affect its truth value.

Circuit Satisfiability:-

The circuit satisfiability involves determining whether there exists an assignment of inputs to a given boolean circuit such that the output of the circuit is true. In other words, it asks whether a given boolean circuit evaluates to true for some combination of input values.

Input:- A boolean circuit C composed of AND, OR and NOT gates, where each gate has fan-in 2 (receives input from two other gates or inputs) and a single output gate.

Output:- Determine whether exists an assignment of inputs to the circuit such that the output of the circuit is true.

Now to prove that Circuit SAT is NP-Complete, we need to show two things:-

1) Circuit SAT is NP: Given a proposed solution (a truth assignment for the inputs), we can verify in polynomial time whether it satisfies the given boolean circuit.

2) Every problem in NP can be reduced to circuit SAT in polynomial time:-

This involves showing that any problem in NP can be transformed to an instance of Circuit SAT in polynomial time.

Proof:

1) Circuit SAT is in NP:-

To verify a solution (truth assignment), we simply need to evaluate the boolean circuit using the given input. Since boolean circuit evaluation can be done in polynomial time (by traversing the circuit in depth-first manner), verifying a solution is also polynomial. Therefore, Circuit SAT is in NP.

2) Every problem in NP can be reduced to circuit SAT in polynomial time:-

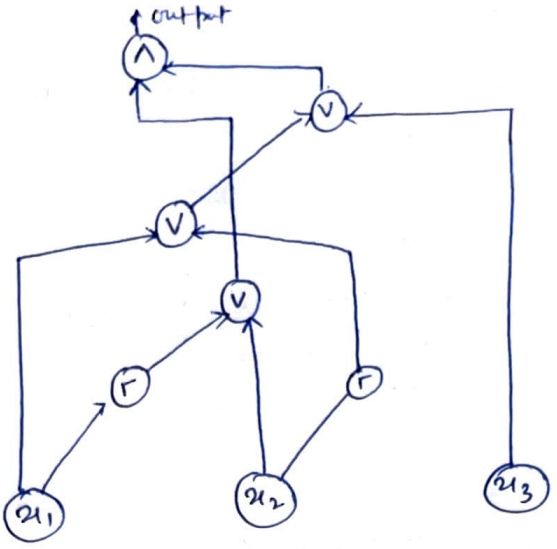
→ Since we are proving Circuit SAT is NP-Complete, we can take any problem in NP and show that it can be reduced to Circuit SAT in polynomial time.

→ Let's take SAT, a classic NP-Complete problem.

→ Given a SAT instance, we can construct an equivalent boolean Circuit by representing each clause in CNF as a series of AND and OR gates. The input to the circuit correspond to the variable in the SAT instance.

→ For Ex: Consider the SAT instance

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2).$$



→ In this circuit, each gate corresponds to an operation (AND or OR), and the inputs to each gate corresponds to the literals in the clauses. The output of the circuit represents whether the entire formula evaluates to true.

→ Therefore, since we can transform any SAT instance into an equivalent boolean circuit in polynomial time, and SAT is NP-complete, Circuit SAT is also NP-Complete.

CHAPTER-9

PSPACE: A class of Problems beyond NP

Definition: The complexity class PSPACE consists of all decision problems that are solvable in polynomial space: i'e an algorithm that uses an amount of space that is polynomial in the size of input.

Memory: No. of different cells the tape head visits.

Theorem: $P \subseteq PSPACE$

Proof: A turing machine that runs for t steps can visit at most t cells.

> Time cannot be reused but space can!

For Ex: An algorithm that just counts from 0 to $2^n - 1$ in base-2 notation. It simply needs to implement an n-bit counter, which it maintains in exactly the same way one increments an odometer in a car.

→ Thus this algorithm runs for an exponential amount of time, and then halts, it has used only polynomial amount of space.

Theorem: $SAT \in PSPACE$

Proof: → Enumerate all truth assignment.

→ Check for each one whether the CNF formula is satisfied. If it is satisfied, return true. If it isn't satisfied for any assignment, return false.

How to do it?

→ We maintain a counter for n variables use n-bit odometer.

only requires polynomial space (n bits)

```
0 0 0
0 0 1
0 1 0
  !
  :
```

In this way we can enumerate all possible truth assignmat through the n variable.

→ Each checks of CNF formula only takes polynomial time, hence it only takes polynomial space.

reuse the same space for each of the 2^n checks.

→ The crucial part here is that once we have checked a particular assignment of variables, plug this into the formula and evaluated the formula, we don't need to remember anything after that, so we can erase everything needed to do to evaluate the formula

from the memory and then reuse that memory for the next truth assignment of the variable.

Corollary: NP ⊆ PSPACE

Proof:- Consider any problem $Y \in NP$. Then $Y \leq_p 3SAT$ (3SAT is NP-complete). So, can decide whether $w \in Y$ by:

i) Computing $f(w)$ ← reduction ← only take polynomial time ⇒ only take polynomial space.

ii) Decide whether $f(w) \in SAT$ ← only takes polynomial space.

→ Both steps in the process requires polynomial space. So, Overall we can decide in polynomial space, whether a problem is in the language Y or not. Therefore this problem Y is also contained in PSPACE.

→ This works for any problem Y in NP and therefore NP is completely contained in PSPACE. **NP ⊆ PSPACE**

Alternative Proof: By using the definition of NP.

Consider any problem $Y \in NP$. Y has certificates of polynomial size and certifier M.

To decide whether $w \in Y$:

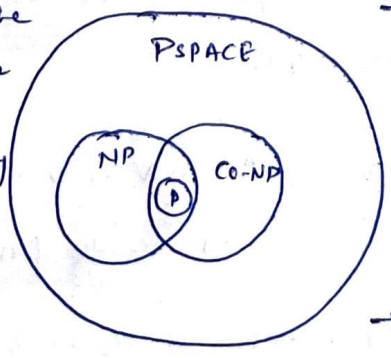
→ Enumerate all bits strings t of polynomial size. ← Use odometer as counter ⇒ Polynomial space.

→ Check for each whether M accepts $\langle w, t \rangle$. If yes, return "yes". If M does not accept for any t, return "no".

Relationship among various classes of Problem:-

→ Just as with the class P, a problem X is in PSPACE iff its complementary problem $\bar{X}$ is in PSPACE.

Hence, co-NP ⊆ PSPACE



→ Given that PSPACE is an enormously large class of problems, containing both NP and co-NP, it is very likely that it contains problems that cannot be solved in P.T.

→ But it has not been proven that $P \neq PSPACE$.

Quantified Satisfiability (QSAT)

The decision problem QSAT:- Given a CNF formula $\phi(x_1, x_2, \ldots x_n)$, decide whether the following is true:

$$\exists x_1 \, \forall x_2 \, \exists x_3 \, \forall x_4 \cdots \exists x_n \, (\phi(x_1, \ldots x_n))$$

↑ Assume that n is odd here.

Interpretation:- Consider a game b/w an existential player Alice and an universal player Bob. Alice picks values for odd variable and Bob picks values for even variable as follows:

Alice picks value for x_1

Bob picks value for x_2

Alice picks value for x_3

⋮

Goal of Alice : Make ϕ true.

Goal of Bob : Make ϕ false.

Does Alice have a winning Strategy?

Ex:

$$(x_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$$

Alice sets x_1 to ~~true~~
- Bob sets x_2 to true – Alice sets x_3 to true.
- Bob sets x_2 to false – Alice sets x_3 to false.

The formula is true either way. So, Alice can win.

* So, this would be a "yes" instance for QSAT.

Example for "No" instance QSAT.

$$(x_1 \vee x_2) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3)$$

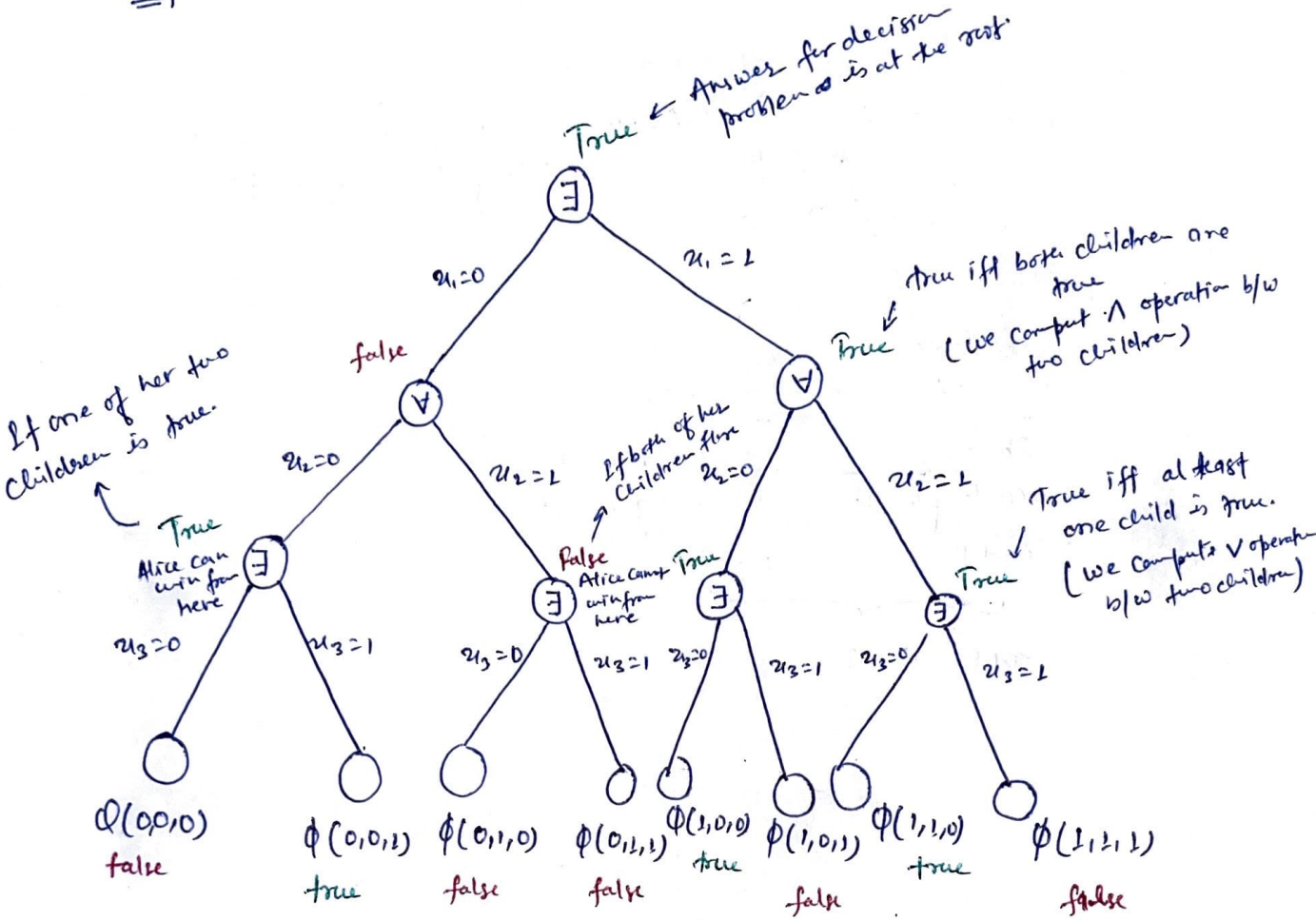
Alice sets x_1 to false – Bob sets x_2 to false → Alice loses since first clause is false.

So, Alice has to sets x_1 to true – Bob sets x_2 to true → Alice loses since either the second or third Clause is false.

QSAT in PSPACE

Theorem: QSAT ∈ PSPACE

Proof.



How we can do this in polynomial space.?

→ The tree has exponential size, so if you just naively evaluate each node of the tree and write all of those evaluation down, we need one bit per node in the tree and there are an exponential number of nodes. So, we would need exponential memory or exponential space. So, this doesn't work, we need a better approach.

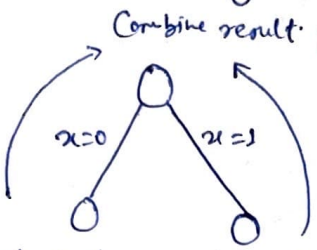
Sol^n: The crucial observation here is that we don't need to memorize all the values at the same time. Once we determined the value of a node in the tree, we can forget about the values of all the children and the children of the children.

It can be claimed that the total space requirement we have is no more than two times the height of the tree, which is the no. of variable in our boolean formula plus the space requirement we have for evaluating the formula for specific truth assignment of the variables.

We only require space $2 \cdot n + C$ ← Space to evaluate ϕ.

↑ Height of tree = No. of variables

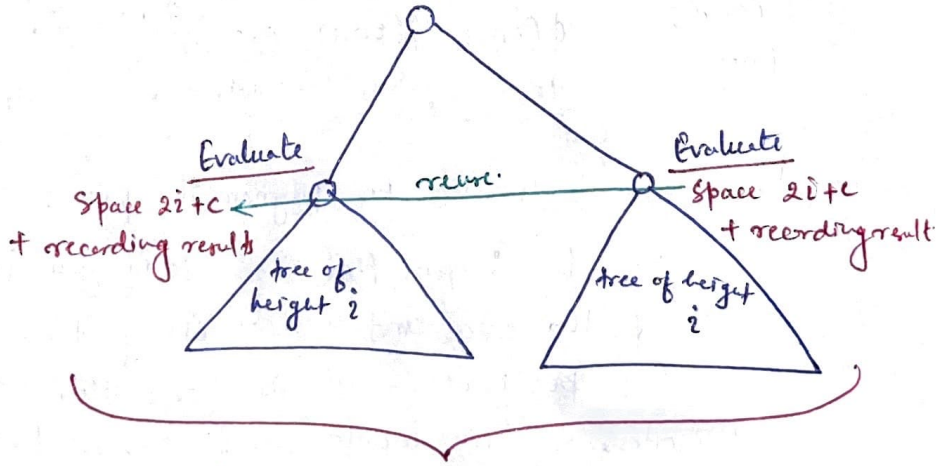
We can prove it by induction.



Evaluate ϕ & record the result — Space C + recording result = $C+1$

Evaluate ϕ & record the result — reuse space + recording result = $C+1$

Total = $C+2$



total = $2 + 2i + c$

$= 2(i+1) + c$

↑

Polynomial space.

Extending the Limits of Tractability

Finding Small Vertex Covers:-

Given a graph $G=(V,E)$ and an integer k, we would like to find a vertex cover of size at most k- that is, a set of nodes $S \subseteq V$ of size $|S| \leq k$, such that every edge $e \in E$ has at least one end in S.

→ Vertex Cover comes with two parameter: n, the number of nodes in the graph, and k, the allowable size of a vertex cover.

→ First of all, we notice that if k is a fixed constant (e.g. $k=2$ or $k=3$) then we can solve vertex cover in polynomial time: we simply try all subsets of V of size k, and see whether any of them constitute a vertex cover.

→ Brute force will take $O(k \cdot n^{k+1})$ time. In which we will try all $C(n,k) = O(n^k)$ subsets of size k.

→ Each subset takes time $O(kn)$ to check whether it is vertex cover.

→ For moderately small values of k, a running time of $O(kn^{k+1})$ is quite impractical.

Eg: If $n=1000$ & $k=10$

It would take 10^{24} seconds to decide if G has a k-node vertex cover. (On a computer executing million high-level instructions per sec.)

→ It turn out that a much better algorithm can be developed, with a running time bound of $O(2^k \cdot kn)$.

Designing the Algorithm:-

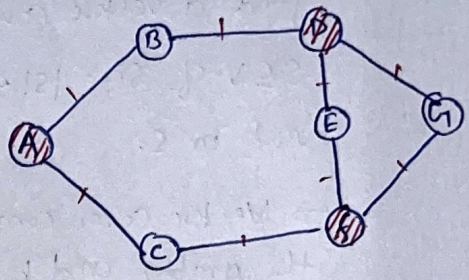
→ As a first observation, we notice that if a graph has a small vertex cover, then it cannot have many edges.

→ Recall that the degree of a node is the number of edges that are incident to it.

Theorem:

If $G = (V, E)$ has n nodes, the maximum degree of any node is at most d, and there is a vertex cover of size at most k, then G has at most kd edges.

Proof: Let S be a vertex cover in G of size $k' \leq k$. Every edge in G has at least one end in S; but each node in S can cover at most d edges. Thus there can be at most $k'd \leq kd$ edges in G.



Vertex Cover = {A, D, F}
Size of VC = 3 → k
d = 3 (Max degree of a node)
n = 7

→ Here, no. of edges are 8 and $kd = 9$, So, the above statement holds for this example.

* Since the degree of any node can be atmost $n-1$, we have the following simple consequence of above theorem:

Theorem: If $G = (V, E)$ has n nodes and a vertex cover of size k, then G has at most $k(n-1) \leq kn$ edges.

Proof:

→ First of all, lets ~~Consider~~ Check that the given graph G contains more than kn edges; if it does, then we know that the answer to the decision problem – Is there a vertex cover of size at most k? is No.

→ From this, we will assume that G contains at most kn edges.

→ Idea behind algorithm:-

→ Begin with considering any edge $e = (u, v)$ in G.

→ In any k-node vertex cover S of G, one of ~~u~~ u or v must belong to S.

→ Suppose u belongs to such a vertex cover S.

→ Then if we delete u and all its incident edges, it must be possible to cover the remaining edges by at most $k-1$ nodes.

→ Defining $G - \{u\}$ to be the graph obtained by deleting u and all its incident edges, there must be a vertex cover of size at most $k-1$ in $G - \{u\}$.

→ Similarly, if v belongs to S, this would imply there is a vertex cover of size at most $k-1$ in $G - \{v\}$.

Formulating the above idea:-

Theorem:- Let $e = (u, v)$ be any edge of G. The graph G has a vertex cover of size at most k iff at least one of the graph $G - \{u\}$ and $G - \{v\}$ has a vertex cover of size at most $k-1$.

Proof:- Suppose, G has a vertex cover S of size at most k. Then S contains at least one of u or v; suppose that it contains u.

→ The set $S - \{u\}$ must cover all edges that have neither end equal to u.

→ Therefore, $S - \{u\}$ is a vertex cover of size at most $k-1$ for the graph $G - \{u\}$.

→ Conversly, suppose that one of $G - \{u\}$ and $G - \{v\}$ has a vertex cover of size at most $k-1$. Suppose in particular that $G - \{u\}$ has such a vertex cover T. Then the set $T \cup \{u\}$ covers all edges in G, so it is a vertex cover for G of size at most k.

Theorem:

The running time of the Vertex Cover algorithm on an n-node graph, with parameter k, in $O(2^k \cdot kn)$.

Proof: We could also prove this by recurrance as follows.

If $T(n, k)$ denotes the running time on an n-node graph with parameter k, then $T(\cdot, \cdot)$ satisfies the following recurrance, for some absolute constant c:

$$T(n, 1) \leq cn,$$

$$T(n, k) \leq 2T(n, k-1) + ckn$$

By induction on $k \geq 1$, it is easy to prove that $T(n,k) \leq c \cdot 2^k kn$.

If this is true for $k-1$, then

$$\begin{aligned} T(n,k) &\leq 2T(n-1, k-1) + ckn \\ &\leq 2c \cdot 2^{k-1}(k-1)n + ckn \\ &= c \cdot 2^k kn - c \cdot 2^k n + ckn \\ &\leq c \cdot 2^k kn. \end{aligned}$$

Approximation Algorithm

Def: An approximation algorithm is a way of dealing with NP-completeness for an optimization problem. This technique doesnot guarantee the best soln. The goal of the approximation algorithm is to come as close as possible to the optimal solution in polynomial time. Such algorithms are called approximation algorithm or heuristic algorithms.

Features of Approximation Algorithm:-

→ An approximation algorithm guarantees to run in polynomial time though it doesnot guarantee the most effective solution.

→ An approximation algorithm guarantees to seek out high accuracy and top quality solution (say within 1% of optimum)

→ Approximation algorithms are used to get an answer near the (optimal) solution of an optimization problem in polynomial time.

Performance ratios of approximation Algorithm:

→ Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution.

→ Depending on the problem, an optimal solution may be defined as one with maximum possible cost or one with minimum possible cost; i/e the problem may be maximization or a minimization problem.

→ Suppose an algorithm for a problem has an approximation ratio of $\rho(n)$ if, for any input size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

→ We call an algorithm that achieve an approximation ratio of $\rho(n)$ a $\rho(n)$-approximation algorithm.

→ The definitions of approximation ratio of $\rho(n)$-approximation algorithm apply for both minimization and maximization problems.

→ For maximization problem:-

$0 < c < c^*$ and the ratio $\frac{c^*}{c}$ gives the factor by which the cost of an optimal solution is larger than the cost of approximate solution.

→ For minimization problem:-

$0 < c^* < c$, and the ratio $\frac{c}{c^*}$ gives the factor by which the cost of approximate solution is larger than the cost of an optimal solution.

→ Since all solutions are assumed to have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $c/c^* < 1$ implies $c^*/c > 1$.

→ What can you do if you need to solve an NP-hard problem?

Solving a problem efficiently means:-

* Solve the problem in polynomial time. — May be all your inputs are small enough so that it doesn't matter.
* Solve arbitrary instance of the problem — May be the problem can be made less general
* Solve the problem to optimality — May be close to optimal is good enough.

Must sacrifies at least one of these ?

Approximation Algo.

Def: An algorithm is an α-approximation algorithm if
* it is guranteed to run in polynomial time.
* it is guranteed to solve arbitrary instance of the problem.
* it is guranteed to find a solution that is within a ratio of α of the optimal solution.

Ex: Maximum independent set of graph has size 100.
→ 2-approximation algorithm gives independent set of size ≥ 50

Maximization Problem:- profit $\geq \frac{1}{2}$ * optimal profit

Minimum vertex cover has size 100.
→ 2-approximation algorithm gives vertex cover of size ≤ 200.

Minimization Problem:-

Cost $\leq \alpha$ * optimal cost.

Load Balancing Problem:-

* m identical machines.
* n jobs
* job j has processing time t_j

also called size or length of the job.

→ A job must run without interruption on a single machine.
→ A single machine can only process one job at a time.

Load of a machine:- How long the machine need to run for to execute all jobs assigned to the machine?

Let J(i) be the subset of jobs assigned to machine i. The load of machine i is $T_i = \sum_{j \in J(i)} t_j$.

Goal:- Assign every job to a machine so that the maximum load $\max\{L_1, L_2 \cdots L_m\}$ is minimized.

Makespan

Designing the algorithm:-

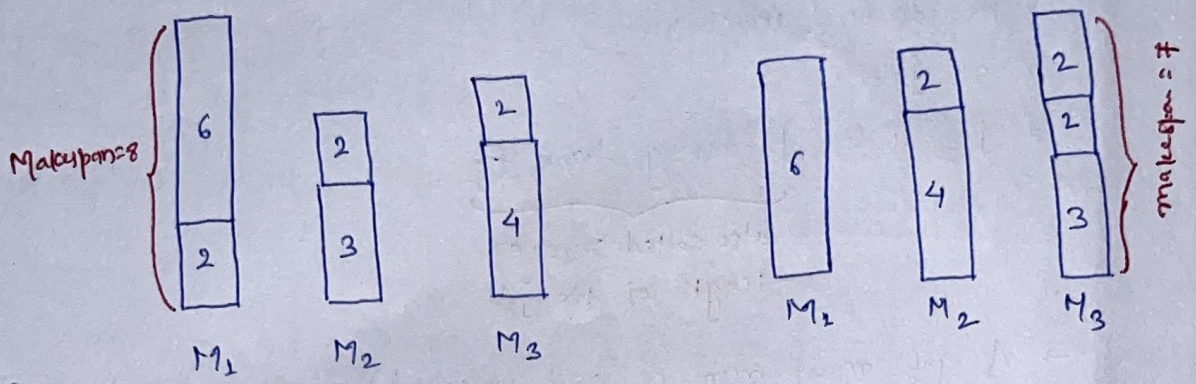
→ First, we consider a very simple greedy algorithm for the problem. The algorithm makes one pass through the jobs in any order; when it comes to job j, it assigns j to the machine whose load is smallest so far.

Algorithm:-

Greedy-Balance:

1. Start with no job assigned
2. Set $T_i = 0$ and $A(i) = \phi$ for all machine M_i
3. For $j = 1 \cdots n$
4. Let M_i be a machine that achieve the minimum $\min_k T_k$
5. Assign job j to machine M_i
6. Set $A(i) \leftarrow A(i) \cup \{j\}$
7. Set $T_i \leftarrow T_i + t_j$
8. End for.

Ex: A sequence of six jobs with size, 2, 3, 4, 6, 2, 2 and three machines M_1, M_2 & M_3.



→ The resulting makespan is 8, the weight of the jobs on the first machine.

→ Note that this is not optimal solution; had the jobs arrived in different order, so that the algorithm saw the sequence of size 6, 4, 3, 2, 2, 2, then it would have produced an allocation with a makespan of 7.

Analyzing the Algorithm:

→ Let T denote the makespan of the resulting assignment; we want to show that T is not much larger than the minimum possible makespan T^*.

→ We need to compare the solution to the optimal value T^*, even though we don't know the value and have no hope to compute it.

→ For the analysis we need a lower bound on the optimum - a quantity with the guarantee that no matter how good the optimum is, it cannot be less than this bound.

→ There are many possible lower bound on the optimum.

→ One idea for a lower bound is based on considering the total processing time $\sum_j t_j$. One of the m machine must do at least a $1/m$ fraction of the total work.

So, the optimal makespan is at least

$$T^* \geq \frac{1}{m} \sum_j t_j \longrightarrow (1)$$

Suppose we have one job that is extremely large relative to the sum of all processing times. In a sufficiently extreme version of this, the optimal soln will place this job on a machine by itself, and it will be the last one to finish.

* This lower bound is still not sufficient to be useful.

So, The optimal makespan is at least $T^* \geq \max_j t_j \longrightarrow (2)$

Lets Consider one situation;

Consider a machine with maximal load T_i. Let j be the last job assigned to that machine.

Load of the machine before job j is assigned is $T_i - t_j \leq T_k$, for all $1 \leq k \leq m$.

Now:

$$\sum_k T_k \geq m \cdot (T_i - t_j)$$

Dividing both side by m

$$\Rightarrow T_i - t_j \leq \frac{1}{m} \sum_k T_k = \frac{\sum_j t_j}{m} \leq T^* \longrightarrow (3)$$

(From 2)

Therefore.

$$T_i = (T_i - t_j) + t_j \leq T^* + T^*$$

(From 3.) (from 2)

$$\leq 2T^*$$

Hence: $\boxed{T_i \leq 2T^*}$

Extensions: An Improved Approximation Algorithm:-

The LPT rule:-

Longest Processing Time (LPT):- Use list scheduling, but consider jobs in decending order of processing time.

Theorem:- LPT is a 1.5 approximation algorithm.

Proof: Assume without loss of generality that $t_1 \geq t_2 \geq t_3 \geq \cdots \geq t_n$.

Lemma. $T^* \geq 2 \cdot t_{m+1} \longrightarrow (4)$

Proof:- Consider the first $m+1$ jobs. Each of these has processing time $\geq t_{m+1}$. Since there are only m machines, two of them must be assigned to the same machine. So, the load of that machine is at least $2 \cdot t_{m+1}$.

Remark:- If there are fewer than $m+1$ jobs, LPT is optimal because each job gets its own machine.

Let machine i be a machine with maximum load T_i. If machine i only contains one job, the solution is optimal. Otherwise, the second job on the machine is not one of the first m jobs in our sequence.

$$T_i = T_i - t_j + t_j$$

$$\leq T^* + t_{m+1}$$

$$\leq T^* + \frac{T^*}{2} \quad \text{(from eqn (4))}$$

$$\boxed{T_i \leq \frac{3}{2} T^*}$$