UNIT - 4

Dynamic Programming

Iterative Improvement

UNIT - 4

Dynamic Programming: Weighted Interval Scheduling: A Recursive Procedure, Subset Sums and Knapsacks: Adding a Variable: The Problem, Designing the Algorithm, Bellman ford Algorithm **(Text book - 1, 6.1, 6.4)**

Dynamic Programming: Warshall's and Floyd's Algorithm. **(Text book - 2, 8.4)**

Iterative Improvement: The Simplex Method, The Maximum-Flow Problem. **(Text book - 2, 10.1, 10.2)**

Weighted Interval Scheduling: A Recursive Procedure

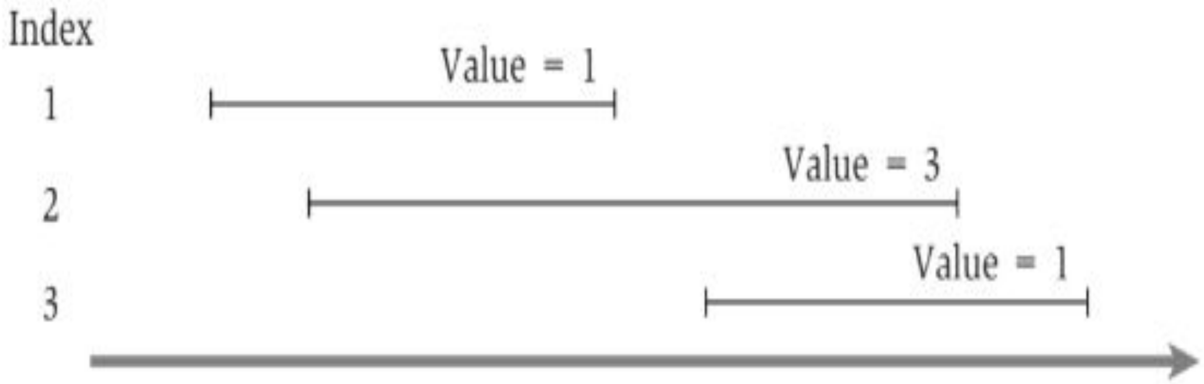


Figure 6.1 A simple instance of weighted interval scheduling.

More Info: https://www.youtube.com/watch?v=K2umaH3vx1Y

Given:

Set of intervals/requests , each interval has a start time , finish time and a value/weight. Two intervals are compatible , if they do not overlap.

Find:

The set of non overlapping intervals such that we can maximize the sum of the values of selected intervals.

$s(i)$ $f(i)$

non-overlapping

greedy strategy

↓

dynamic programming

$a = 1000$

$b = 100$

$f(i)$

α

select the interval that have maximum weight

2

$a = 2$

$b = 1$ $c = 1$ $d = 1$

~~ok~~ ~~ok~~ ~~ok~~

3

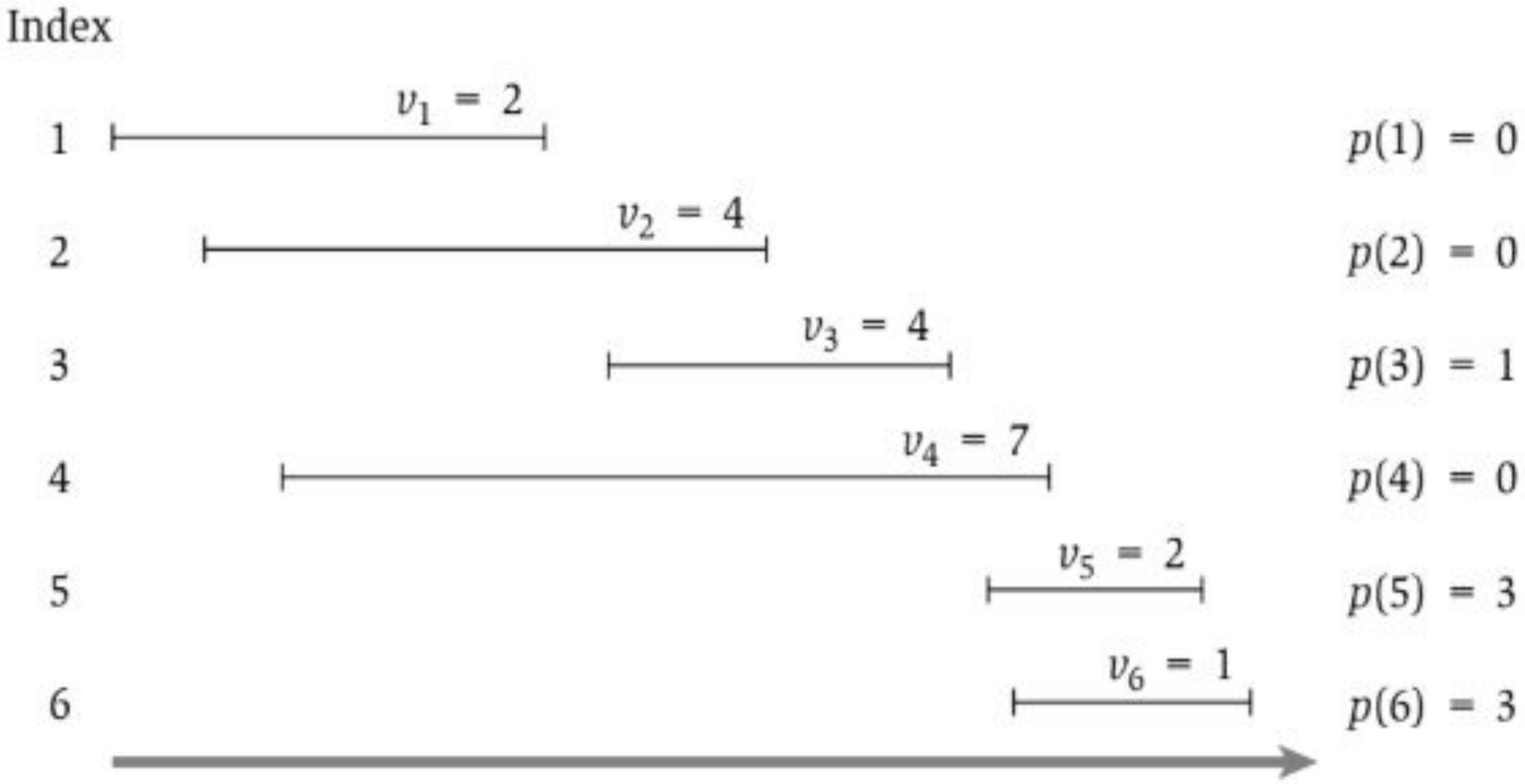
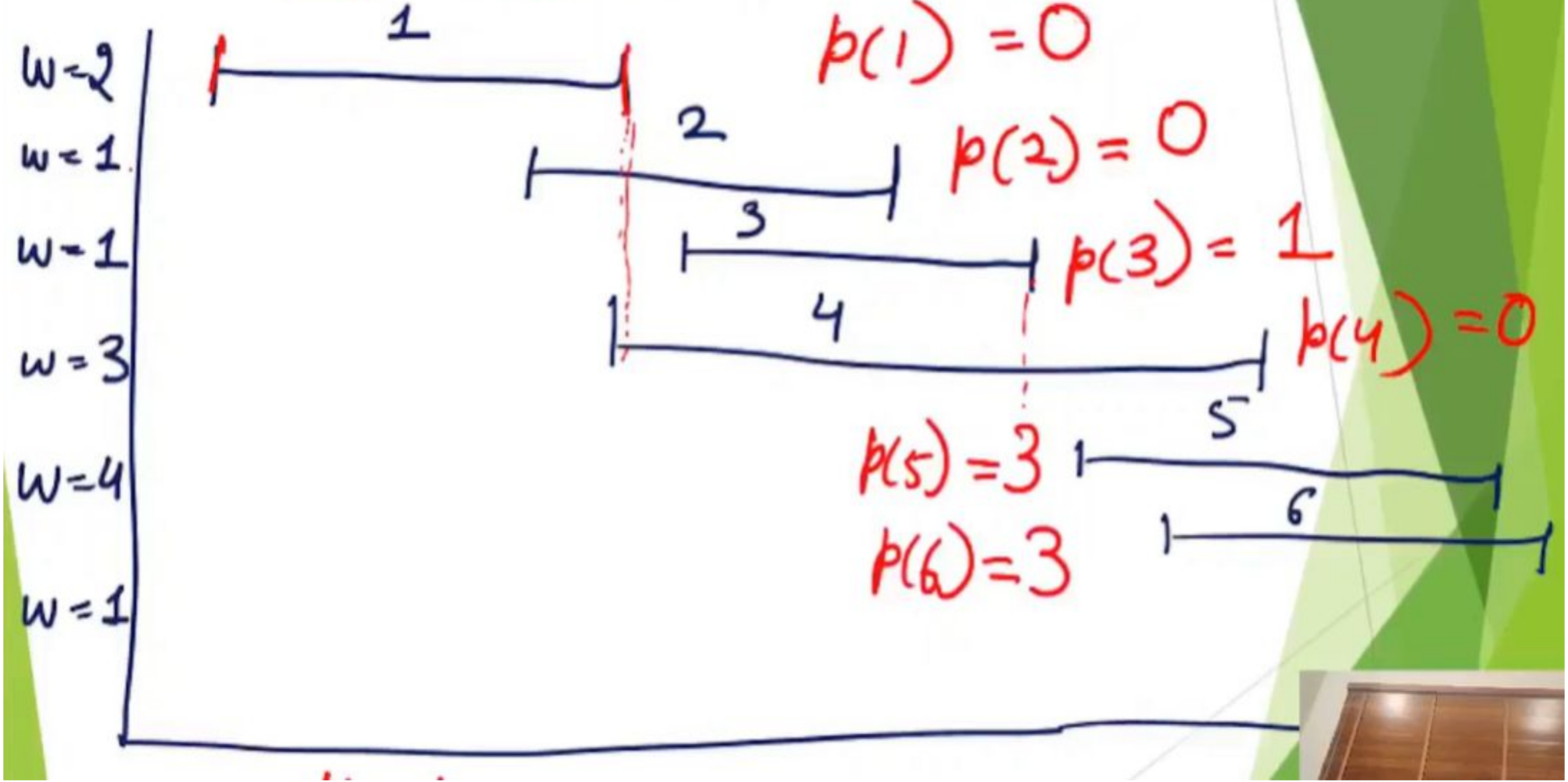


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

For each interval j , compute p(j) largest index i < j such that i and j are disjoint



$\{1, 2 \cdots\cdots 6\}$

nth interval

Consider optimal solution

O

$n \in O$

no job b/w $p(n)$ to n will $\notin O$

1 to $p(n)$ to be part of our optimal

$n \notin O$

This means

O consists of job from $\{1 \text{ to } n-1\}$

This is our recursive statement for finding the optimal solution of intervals

$[1, 2 \cdots n]$

↓

finding the optimal solution for the smaller problems i.e. $[1 \cdots j]$

Compute-OPT(j)

{

If $j = 0$

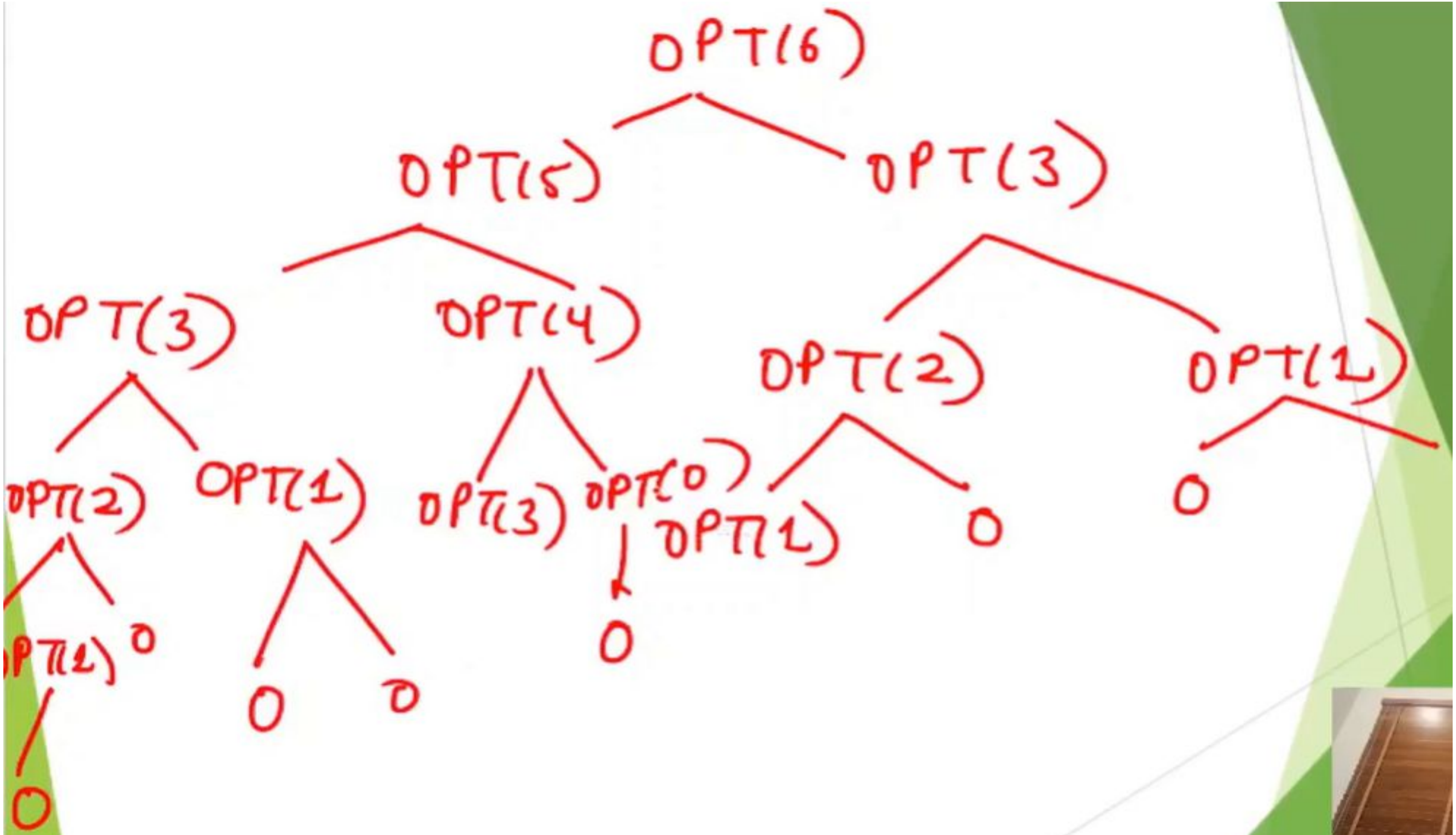
Return 0

else

Return max(v_j + Compute-OPT(p(j)),

Compute-OPT(j − 1))

}


OPT(6)
OPT(5)
OPT(3)
OPT(3)
OPT(4)
OPT(2)
OPT(1)
OPT(2)
OPT(1)
OPT(3)
OPT(0)
OPT(1)
0
0
OPT(1)
0
0
0
0
0
0

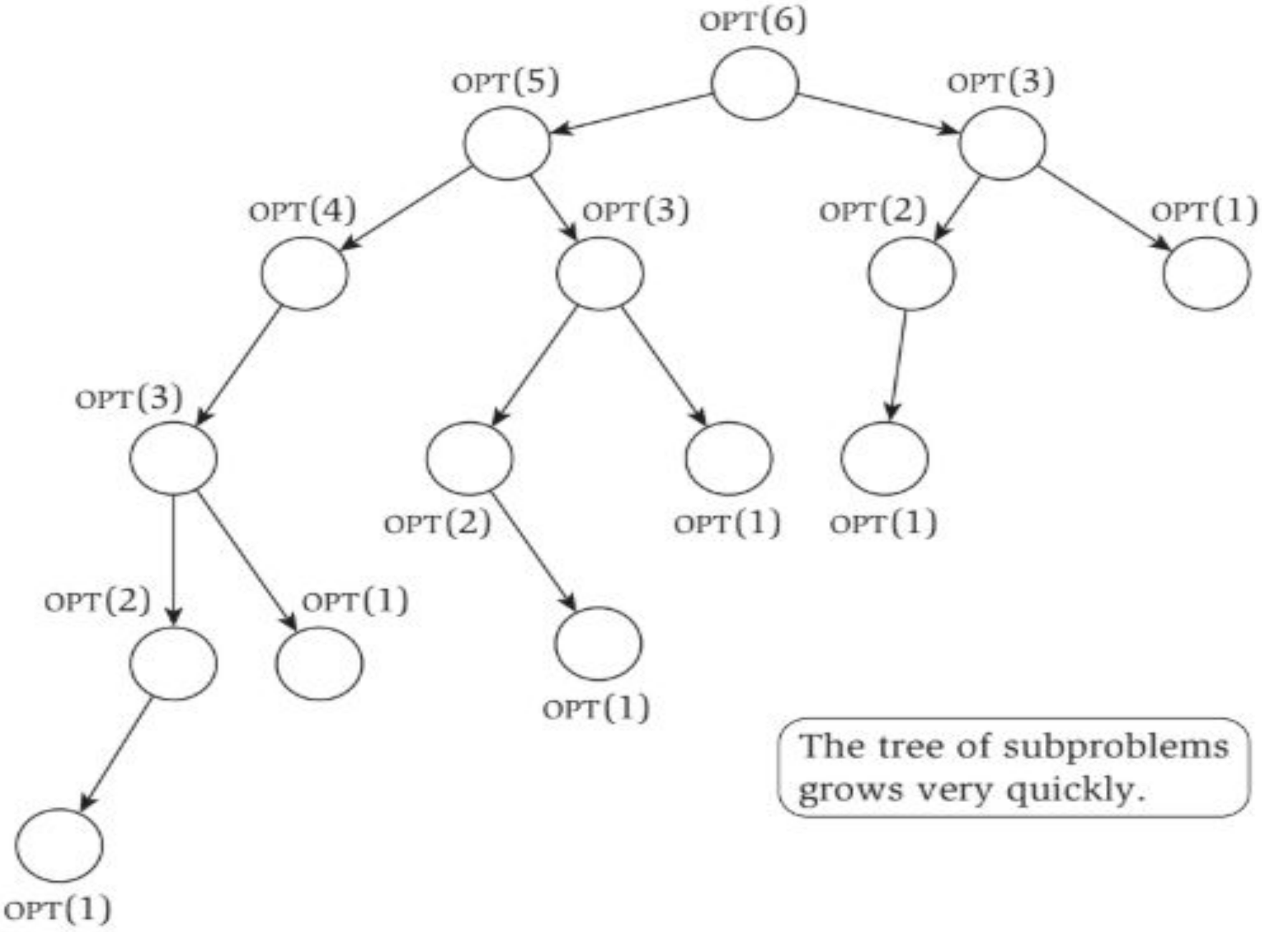


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

Subset Sums and Knapsacks: Adding a Variable

The Problem

In the scheduling problem we consider here, we have a single machine that can process jobs, and we have a set of requests {1, 2, . . . , n}. We are only able to use this resource for the period between time 0 and time W, for some number W. Each request corresponds to a job that requires time wi to process. If our goal is to process jobs so as to keep the machine as busy as possible up to the “cut-off” W, which jobs should we choose?

- This problem is a natural special case of a more general problem called the **Knapsack Problem**, where each request i has both a value vi and a weight wi.
- The goal in this more general problem is to select a **subset of maximum total value,** subject to the restriction that its total weight not exceed W.

Given a set of n items with weights $w_1, w_2 \dots\dots\dots w_n$. Choose a subset S to

$n_1 \quad n_2 \quad n_3 \quad \dots \quad n_n$

$w_1 \quad w_2 \quad w_3 \quad \dots \quad w_n$

maximize

$$\sum_{i \in S} w_i \leq W$$

under the constraint

n_1
$w_1 = 2$

n_2
$w_2 = 2$

n_3
$w_3 = 3$

$W = 6.$

n items

2^n Generate all 2^n subsets of n items, calculate weight & choose max

$w_1 = 2$ $\qquad w_2 = 2$ $\qquad w_3 = 3$

$\boxed{W = 6}$

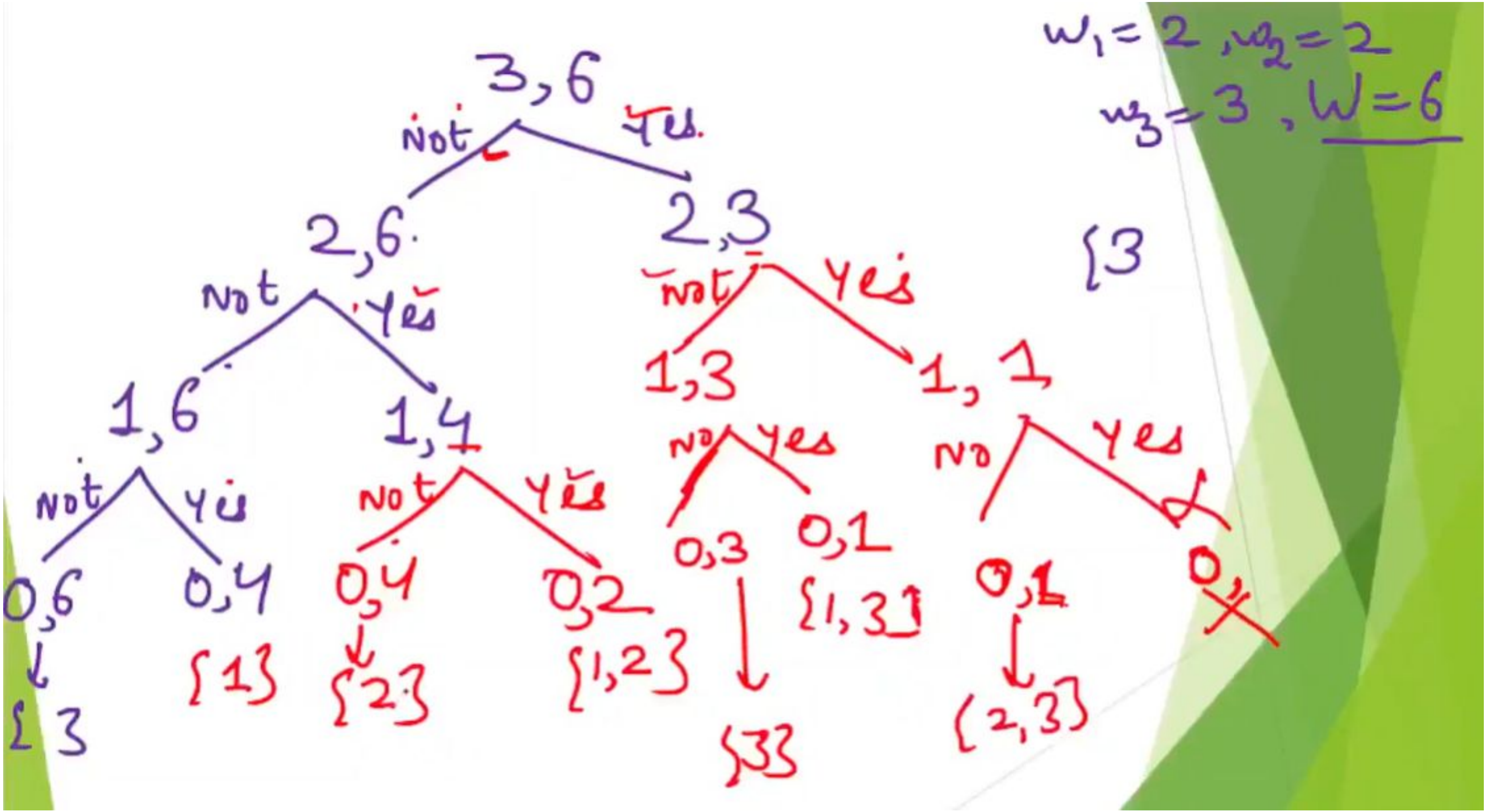
$\{\}, \underset{\uparrow}{\{1\}}^{2}, \underset{\uparrow}{\{2\}}^{2}, \underset{\uparrow}{\{3\}}^{3}, \{1,2\}, \{2,3\}, \{1,3\},$

$\{1,2,3\} \to 7$ ✗

$\{1,2\} \to 4$

$\{2,3\} \to 5$

$\{1,3\} \to 5$



w1 = 2, w2 = 2
w3 = 3, W = 6
3,6
Not
Yes
2,6
2,3
{3
Not
Yes
Not
Yes
1,6
1,4
1,3
1,1
Not
Yes
Not
Yes
No
yes
No
yes
0,6
0,4
0,4
0,2
0,3
0,1
0,1
0,x
{3
{1}
{2}
{1,2}
{3}
{1,3}
{2,3}

Q For subproblem for items $(1 \ldots\ldots i)$ and maximum allowed weight w, give the recurrence.

$6 < 8$ w_i

$\{1 \ldots\ldots i\}$

$w < w_i$ then $OPT(i, w) = OPT(i-1, w)$

otherwise

$$OPT(i, w) = \max\left(OPT(i-1, w), w_i + OPT(i-1, w-w_i)\right)$$

Designing the Algorithm

```
Subset-Sum(j, W)
 {
  if (j = = 0)   //empty set
   return 0
 else if ( W < wj)
  return  subset-sum(j-1, W)

 else
 return  max( subset-sum(j-1, w),
              wj +  sub-s -sum((j-1, W-wj))
```

$W = 5$

$w_1 = 1, w_2 = 2, w_3 = 3$

n+1 columns

```
Subset-Sum (n , W)
{
array M[0......................n,0..................W]

Initialize M[0,W] = 0 for each w=0,1,2.............W

For(i=1,2......................n)
  For(w = 0,1,2................W)
{
      if(W < wi )
      M[i,w] = M[i-1,w]
      else
       M[i,w] = max( M[i-1,w] , wi + M[i-1, W - wi  ]
}
Return M[n,W]
}
```

n=3	0	1	2	3	4	5
n=2	0	1	2	3	3	3
n=1	0	1	1	1	1	1
n=0	0	0	0	0	0	0
	0	1	2	3	4	5

we want W

max

$M[3,4] = \max(M[2,4], 3 + M[2,1])$

$= \max(3, 3 + 1)$

$M[3,5] = \max(M[2,5], 3 + M[2,2])$

$3, 3+2$

Analyzing the Algorithm

Knapsack size $W = 6$, items $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$

3	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

Figure 6.12 The iterations of the algorithm on a sample instance of the Subset Sum Problem.

- If $n \notin \mathcal{O}$, then $\text{OPT}(n, W) = \text{OPT}(n-1, W)$.
- If $n \in \mathcal{O}$, then $\text{OPT}(n, W) = v_n + \text{OPT}(n-1, W - w_n)$.

Using this line of argument for the subproblems implies the following analogue of (6.8).

(6.11) *If $w < w_i$ then $\text{OPT}(i, w) = \text{OPT}(i-1, w)$. Otherwise*

$$\text{OPT}(i, w) = \max(\text{OPT}(i-1, w), v_i + \text{OPT}(i-1, w - w_i)).$$

Using this recurrence, we can write down a completely analogous dynamic programming algorithm, and this implies the following fact.

(6.12) *The Knapsack Problem can be solved in $O(nW)$ time.*

Running time $\longrightarrow$ proportional to the no. of entries in the table

Subset-sum

$\downarrow$ $(n+1, W+1)$

Subset sum algorithm computes the optimal value of the problem

0/1 Knapsack Problem

Given **n** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all

For more Info:

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Fractional Knapsack Problem

Given two arrays, **val[]** and **wt[]**, representing values and weights of items, and an integer **capacity** representing the maximum weight a knapsack can hold, the task is to determine the **maximum total value** that can be achieved by putting items in the knapsack. You are allowed to break items into fractions if necessary.

For more Info: https://www.geeksforgeeks.org/fractional-knapsack-problem/

Ex: Consider the example: val[] = [60, 100, 120], wt[] = [10, 20, 30], capacity = 50. Store the value and weight of each item in form {value, weight}. **Sorting:** Initially sort the array based on the profit/weight ratio. The sorted array will be **{{60, 10}, {100, 20}, {120, 30}}**.

- For **i = 0**, weight = 10 which is less than capacity. So add this element. **profit = 60** and remaining **capacity = 50 - 10 = 40**.
- For **i = 1**, weight = 20 which is less than capacity. So add this element too. **profit = 60 + 100 = 160** and remaining **capacity = 40 - 20 = 20**.
- For **i = 2**, weight = 30 is greater than capacity. So add 20/30 fraction = **2/3** fraction of the element. Therefore **profit** = 2/3 * 120 + 160 = 80 + 160 = **240** and remaining **capacity** becomes **0**. So the final profit becomes **240** for **capacity = 50**.

Step by step approach:

1. Calculate the ratio (**profit/weight**) for each item.
2. Sort all the items in decreasing order of the ratio.
3. Initialize **res = 0**, current capacity= given capacity.
4. Do the following for every item **i** in the sorted order:

   a) If the weight of the current item is less than or equal to remaining capacity then add the value of that item into **result**

   b) Else add the current item as much as we can and break out of the loop.
5. Return **res**.

Sr. No	0/1 knapsack problem	Fractional knapsack problem
1.	The 0/1 knapsack problem is solved using dynamic programming approach.	Fractional knapsack problem is solved using a greedy approach.
2.	In the 0/1 knapsack problem, we are not allowed to break items.	Fractional knapsack problem, we can break items for maximizing the total value of the knapsack.
3.	0/1 knapsack problem, finds a most valuable subset item with a total value less than equal to weight.	In the fractional knapsack problem, finds a most valuable subset item with a total value equal to the weight if the total weight of items is more than or equal to the knapsack capacity.
4.	In the 0/1 knapsack problem we can take objects in an integer value.	In the fractional knapsack problem, we can take objects in fractions in floating points.

Bellman Ford Algorithm

Given a weighted graph with **V** vertices and **E** edges, along with a source vertex **src**, the task is to compute the shortest distances from the source to all other vertices. If a vertex is unreachable from the source, its distance should be marked as **10^8**. In the presence of a negative weight cycle, **return -1** to signify that shortest path calculations are not feasible.

- To get started thinking about the algorithm, we begin by adopting the original version of the Bellman-Ford Algorithm, which was less efficient in its use of space. We first extend the definitions of OPT(i, v) from the Bellman-Ford Algorithm, defining them for values $i \geq n$.
- **Bellman-Ford** is a **single source** shortest path algorithm. It effectively works in the cases of negative edges and is able to detect negative cycles as well. It works on the principle of **relaxation of the edges**.

Practice problem on Bellman Ford Algorithm

b) Find the shortest path from node 1 to every other node in the given graph using Bellman-Ford algorithm.

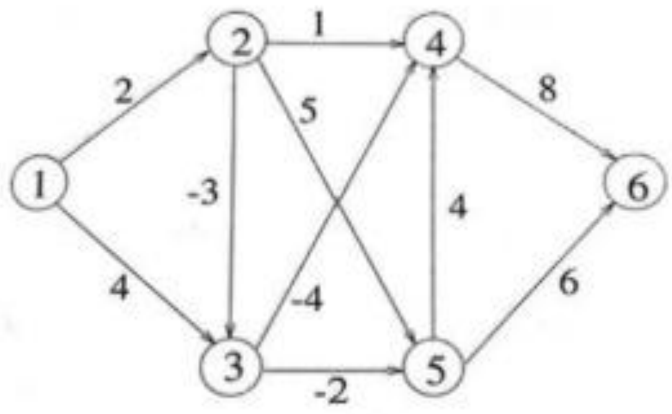


Fig.7(b)

Warshall’s and Floyd’s Algorithms

Warshall’s algorithm for computing the transitive closure of a **directed graph** and Floyd’s algorithm for the all-pairs shortest-paths problem.

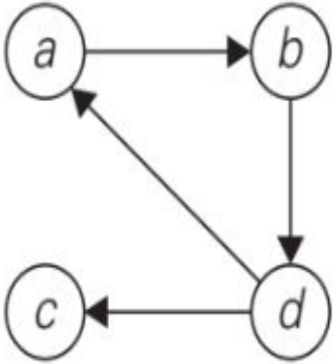
DEFINITION The transitive closure of a directed graph with n vertices can be defined as the **n × n** boolean matrix T = {tij }, in which the element in the ith row and the j th column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the j th vertex; otherwise, tij is 0.

ALGORITHM *Warshall*$(A[1..n, 1..n])$

//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

Since this method traverses the same digraph several times, we should hope that a better algorithm can be found. Indeed, such an algorithm exists. It is called **Warshall's algorithm.**



(a)

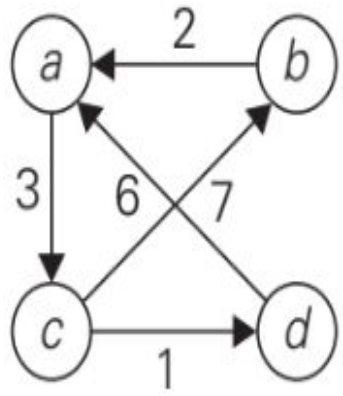
$$A = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b)

$$T = \begin{array}{c c} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.11 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.



(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.14 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

ALGORITHM *Floyd*($W[1..n, 1..n]$)

//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

For more Info:

https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

- The graph **may contain negative edge weights**, but it **does not contain any negative weight cycles**.
- This algorithm works for both the **directed** and **undirected weighted** graphs and can handle graphs with both **positive** and **negative weight edges**.

Note: It does not work for the graphs with **negative cycles** (where the sum of the edges in a cycle is negative).

No matter how many edges are there in the graph the **Floyd Warshall Algorithm** runs for O(V3) times

Warshall’s and Floyd’s Practice Problem

Solve the all-pairs shortest-path problem for the digraph with the following weight matrix:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

Step 1: Initial Weight Matrix

Given matrix (from Fig. 7(b)):

$$W = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

We denote this matrix as `D(0)`, the initial distance matrix. Here, `∞` (infinity) represents no direct path between nodes.

Step 2: Floyd-Warshall Algorithm

$$D^{(0)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \infty & 4 & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ \infty & \infty & 0 & 4 & 7 \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

$$D^{(5)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ 10 & 12 & 0 & 4 & 7 \\ 7 & 9 & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

Step 3: Floyd-Warshall Algorithm

The algorithm updates the distance matrix by checking for all intermediate vertices `k` from `0` to `n-1` (n = 5 in this case). For each pair `(i, j)`, update:

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j],\ D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

We'll iteratively apply this from `k = 0` to `4`.

To save space, I'll show the final result after all iterations.

Final Distance Matrix (All-Pairs Shortest Paths):

$$D = \begin{bmatrix} 0 & 2 & 5 & 1 & 4 \\ 5 & 0 & 3 & 2 & 5 \\ 10 & 12 & 0 & 4 & 7 \\ 7 & 9 & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

The Simplex Method

Geometric Interpretation of Linear Programming

- **Linear programming** is a mathematical concept that is used to find the optimal solution of the linear function.
- **Linear programming** is the technique used for optimizing a particular scenario. Using linear programming provides us with the best possible outcome in a given situation.

How to Solve Linear Programming Problems?

Step 1: Mark the decision variables in the problem.

Step 2: Build the objective function of the problem and check if the function needs to be minimized or maximized.

Step 3: Write down all the constraints of the linear problems.

Step 4: Ensure non-negative restrictions of decision variables.

Step 5: Now solve LPP using any method generally we use either the simplex or graphical method.

An Outline of the Simplex Method

- It must be a maximization problem.
- All the constraints (except the nonnegativity constraints) must be in the form of linear equations with nonnegative right-hand sides.
- All the variables must be required to be nonnegative.

Solve following LPP Geometrically

a) maximize $3x + y$

subject to $-x + y \leq 1$

$2x + y \leq 4$

$x \geq 0, y \geq 0$

b) maximize $x + 2y$

subject to $4x \geq y$

$y \leq 3 + x$

$x \geq 0, y \geq 0$

Practice Problem on LPP (PYQ - 2023)

Solve the following linear programming problems.

Maximize $3x+y$

Subject to $-x+y\leq 1$

$2x+y\leq 4$

$x\geq 0,\ y\geq 0$

The Maximum-Flow Problem

- we consider the important problem of maximizing the flow of a material through a transportation network (pipeline system, communication system, electrical distribution system, and so on).
- We will assume that the transportation network in question can be represented by a connected weighted digraph with n vertices numbered from 1 to n and a set of edges E,

Properties

- It contains exactly one vertex with no entering edges, this vertex is called **source** and assumed to be numbered **1**.
- It contains exactly one vertex with no leaving edges, this vertex is called **sink** and assumed to be numbered **n**.
- The weight uij of each directed edge (i, j) is a positive integer, called the **edge capacity.**

Maximum Flow Problem

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices Source S and sink T in the graph Find out the maximum possible flow from S to T with following constraints.

a) Flow on an edge doesn't exceed given capacity of edge.

b) In-flow is equal to Out-flow for every vertex except s and t

Ford-Fulkerson Algorithm

The following is a simple idea of the algorithm

1) Start with a initial flow as **0**.

2) While there is an **augmenting path** from **source** to **sink**

Add this path flow to flow

3) Return flow

Terminologies

Residual Graph: It's a graph which indicates additional possible flow. If there is such path from Source to sink then there is a possibility to add flow

Residual Capacity: It's original capacity of Flow edge minus flow. **Minimal cut:** Also Known as bottleneck capacity, which decides maximum possible flow from Source to sink through an augmented path

Augmenting path: Augmenting path can be done in 2 ways -

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

Ford-Fulkerson method / augmenting-path method

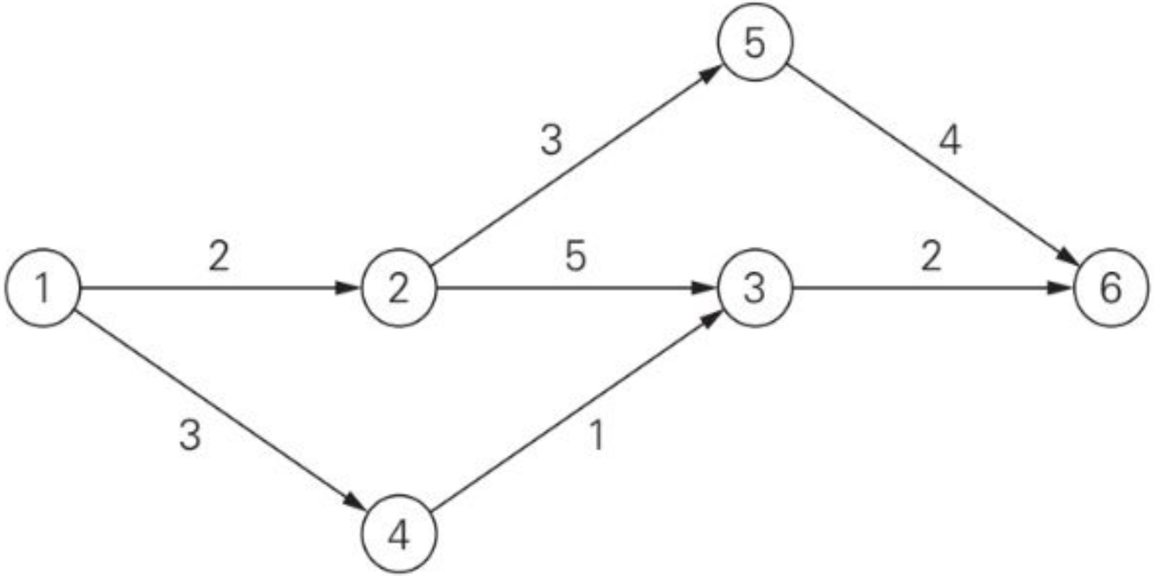


FIGURE 10.4 Example of a network graph. The vertex numbers are vertex "names"; the edge numbers are edge capacities.

Shortest Augmenting Path Algorithm

ALGORITHM *ShortestAugmentingPath(G)*

//Implements the shortest-augmenting-path algorithm
//Input: A network with single source 1, single sink n, and
// positive integer capacities u_{ij} on its edges (i, j)
//Output: A maximum flow x
assign $x_{ij} = 0$ to every edge (i, j) in the network
label the source with ∞, – and add the source to the empty queue Q

```
while not Empty(Q) do
    i ← Front(Q);   Dequeue(Q)
    for every edge from i to j do   //forward edges
        if j is unlabeled
            r_ij ← u_ij − x_ij
            if r_ij > 0
                l_j ← min{l_i, r_ij};   label j with l_j, i⁺
                Enqueue(Q, j)
    for every edge from j to i do   //backward edges
        if j is unlabeled
            if x_ji > 0
                l_j ← min{l_i, x_ji};   label j with l_j, i⁻
                Enqueue(Q, j)
```

if the sink has been labeled

    //augment along the augmenting path found

    $j \leftarrow n$ //start at the sink and move backwards using second labels

    while $j \neq 1$ //the source hasn't been reached

        if the second label of vertex j is i^+

            $x_{ij} \leftarrow x_{ij} + l_n$

        else //the second label of vertex j is i^-

            $x_{ji} \leftarrow x_{ji} - l_n$

        $j \leftarrow i;$ $i \leftarrow$ the vertex indicated by i's second label

    erase all vertex labels except the ones of the source

    reinitialize Q with the source

return x //the current flow is maximum