Module 4
Dynamic Programming

Multistage graphs

Sudeep Manohar
Department of Information Science and Engineering
JNNCE, Shivamogga

Dynamic Programming

- Used for optimization problems either to find the maximum or minimum optimum result
- Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions

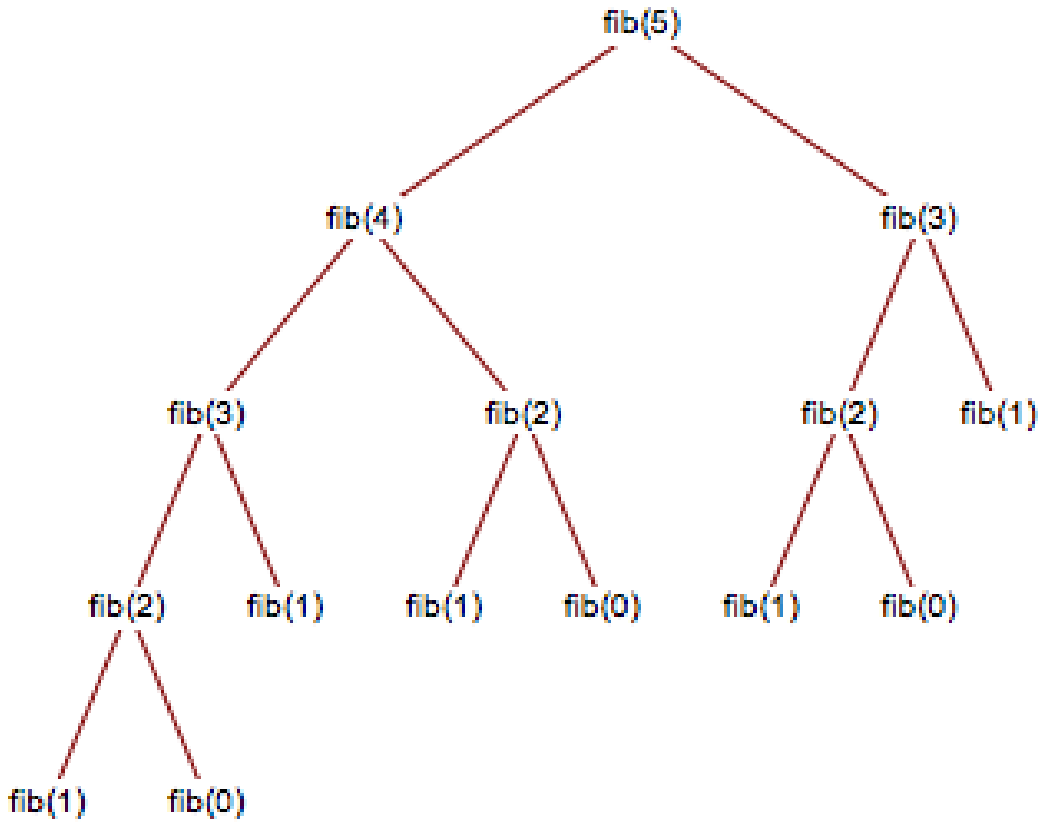
Examples

- Knapsack: We have to decide the values of x_i. An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$

- Shortest path: To find shortest path from vertex i to j, we should decide which vertex is second, which is third and so on until j is reached. An optimal sequence of decisions is one that results in shortest path
- An optimal sequence of decisions can be found by making decisions one at a time and never making an erroneous decision. Greedy method works on this principle
- But for many problems it is not possible to make stepwise decisions
- For such problems we have to try out all possible decision sequences and have to pick the best
- But this consumes more time and space

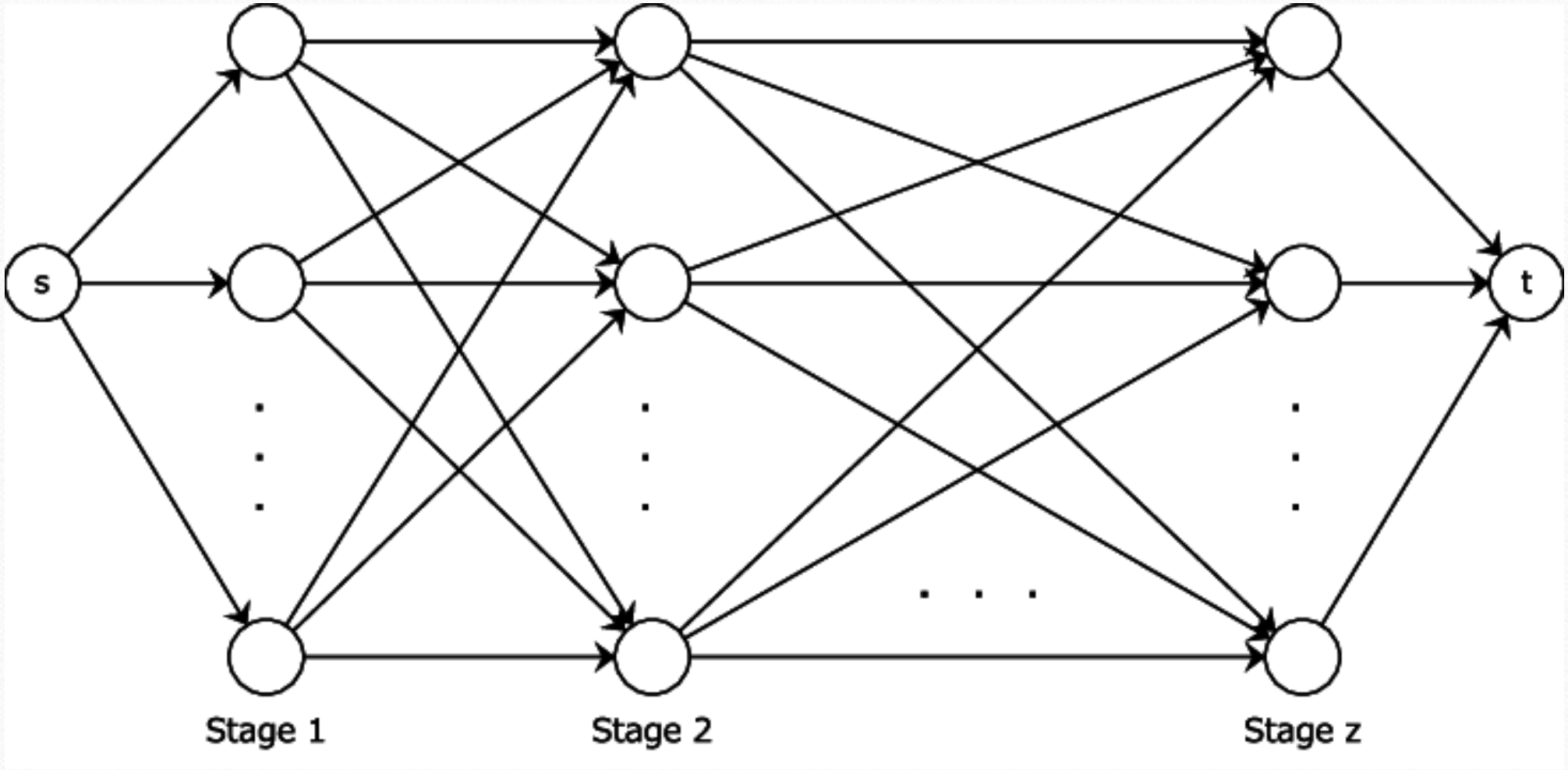
- Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumerations of some decision sequences that cannot possibly be optimal and this is based on principle of optimality
- *Principle of optimality* - The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- In Greedy method there will be one decision sequence but in dynamic programming many decision sequences may be generated

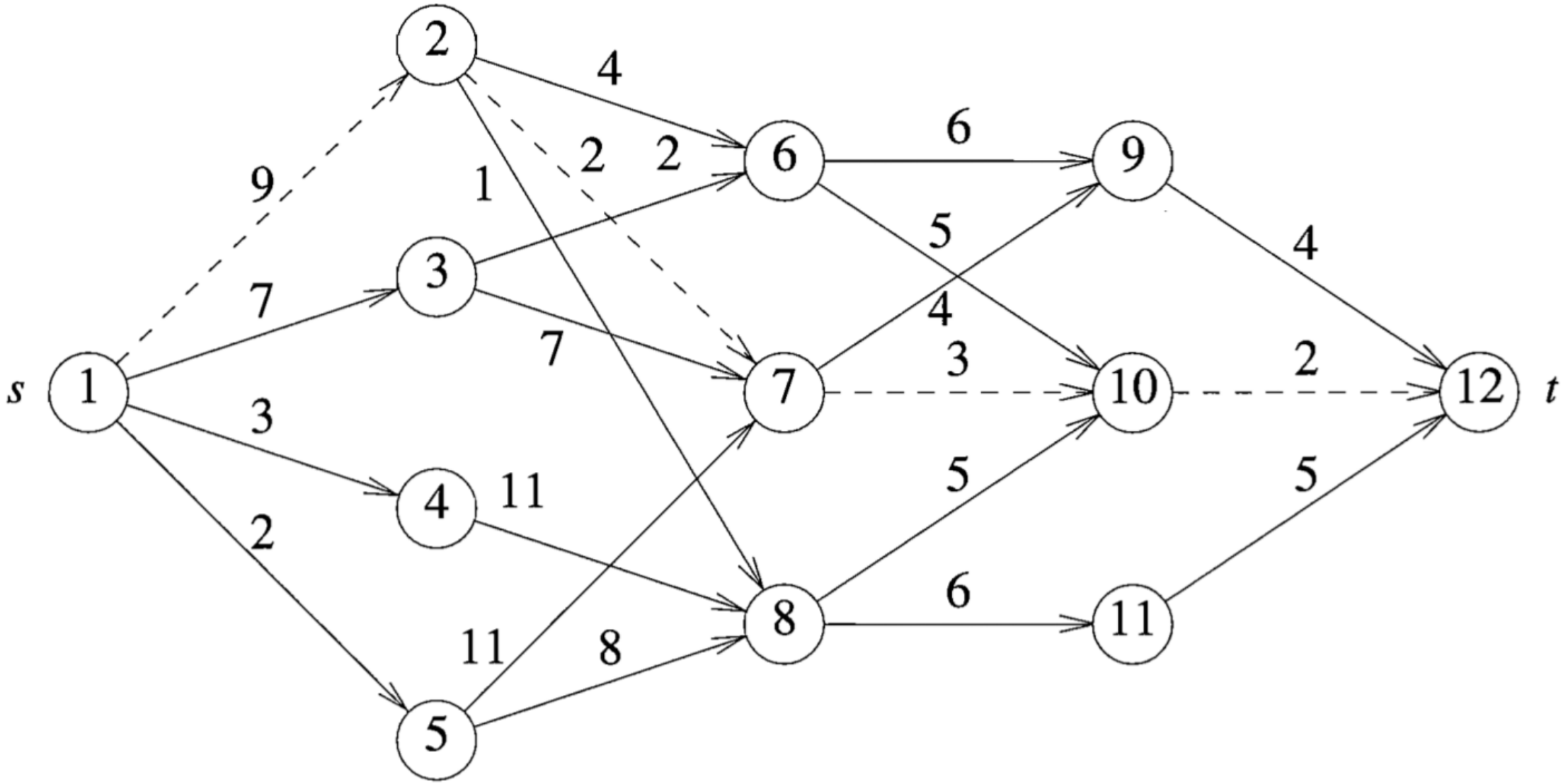
- Dynamic programming design technique is computationally efficient as it stores the intermediate results which can be used for further computation
- Re-computation of the same subproblem is avoided



Multistage Graphs

- A multi stage graph G = (V, E) is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1<i < k$.
- In addition, if $\langle u, v \rangle$ is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1< i < k$.
- The setsV_1 and V_k are such that $|V_1| = |V_k| = 1$.
- Let s and t, respectively, be the vertices in V_1 and V_k.
- The vertex s is the source, and t the sink.
- Let c(i, j) be the cost of edge $\langle i, j \rangle$.
- The cost of a path from s to t is the sum of the costs of the edges on the path.
- The multistage graph problem is to find a minimum-cost path from s to t
- Each set Vi defines a stage in the graph


s
t
Stage 1
Stage 2
Stage z


s
1
2
3
4
5
6
7
8
9
10
11
12
t
9
4
1
2
2
6
7
7
5
4
4
3
2
3
11
2
11
8
6
5
5

$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\}$$

cost(5, 12) = 0

cost(4, 9) = min{ c(9, 12) + cost(5, 12)} = {4+0} = 4

cost(4, 10) = min{ c(10, 12) + cost(5, 12)} = {2+0} = 2

cost(4, 11) = min{ c(11, 12) + cost(5, 12)} = {5+0} = 5

cost(3, 6) = min{ c(6, 9) + cost(4, 9), c(6, 10) + cost(4, 10)}

min{ 6+4, 5+2} = min{ 10, 7} = 7

cost(3, 7) = min{ c(7, 9) + cost(4, 9), c(7, 10)+cost(4, 10)}

= min{ 4+4, 3+2} = min{8, 5} = 5

cost(3, 8) = min{ c(8, 10) + cost(4, 10), c(8, 11) + cost(4, 11)}

= min{ 5+2, 6+5} = min{7, 11} = 7

$$\text{cost}(2, 2) = \min\{ c(2, 6) + \text{cost}(3, 6), c(2, 7) + \text{cost}(3, 7), c(2, 8) + \text{cost}(3, 8)\}$$

$$= \min\{ 4+7, 2+5, 1+7\} = \min\{11, 7, 8\} = 7$$

$$\text{cost}(2, 3) = \min\{ c(3, 6) + \text{cost}(3, 6), c(3, 7) + \text{cost}(3, 7)\}$$

$$= \min\{ 2+7, 7+5\} = \min\{9, 12\} = 9$$

$$\text{cost}(2, 4) = \min\{ c(4, 8) + \text{cost}(3, 8)\}$$

$$= \min\{ 11+7\} = 18$$

$$\text{cost}(2, 5) = \min\{ c(5, 7) + \text{cost}(3, 7), c(5, 8) + \text{cost}(3, 8)\}$$

$$= \min\{ 11+5, 8+7\} = \min\{16, 15\} = 15$$

$$\text{cost}(1, 1) = \min\{ c(1, 2) + \text{cost}(2, 2), c(1, 3) + \text{cost}(2, 3), c(1, 4) + \text{cost}(2, 4), c(1, 5) + \text{cost}(2, 5)\}$$

$$= \min\{ 9+7, 7+9, 3+18, 2+15 \} = \min\{ 16, 16, 21, 17\} = 16$$

Let d(i, j) be the value of l (where l is a node) that minimizes c(j, l)+ cost(i+1, l)

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	-

Shortest paths

Path 1 - p

1	2	7	10	12

minimum cost = 16

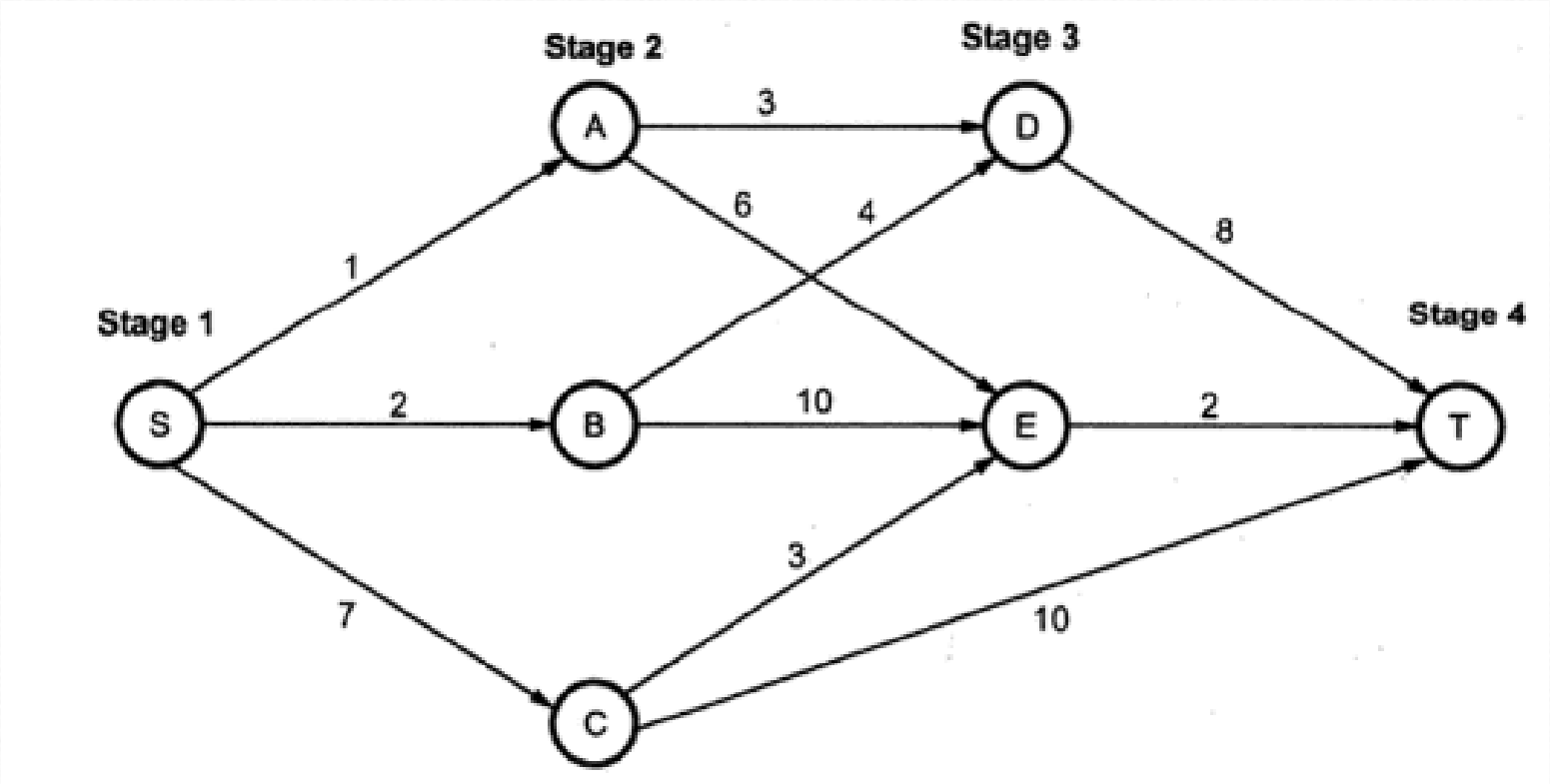
Path 2 - p

1	3	6	10	12

minimum cost = 16

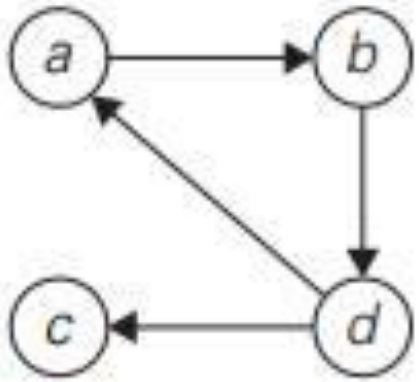
```
Algorithm FGraph(G, k, n, p)
// The input is a k-stage graph G = (V, E) with n vertices
// indexed in order of stages. E is a set of edges and c[i, j]
// is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
{
     cost[n] := 0.0;
     for j := n − 1 to 1 step −1 do
     { // Compute cost[j].
          Let r be a vertex such that ⟨j, r⟩ is an edge
          of G and c[j, r] + cost[r] is minimum;
          cost[j] := c[j, r] + cost[r];
          d[j] := r;
     }
     // Find a minimum-cost path.
     p[1] := 1; p[k] := n;
     for j := 2 to k − 1 do p[j] := d[p[j − 1]];
}
```

Todo



Warshall Algorithm

- It is used to compute the Transitive closure of a directed graph
- Let A = $\{a_{ij}\}$ be the adjacency matrix of a directed graph
- The transitive closure of a directed graph with n vertices can be defined as the n x n Boolean matrix T = $\{t_{ij}\}$, in which the elements in the i^{th} row and the j^{th} column is 1 if there exists a non trivial directed path from the i^{th} vertex to the j^{th} vertex, otherwise t_{ij} is 0



(a)

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

(b)

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

(c)

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

Rules

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$

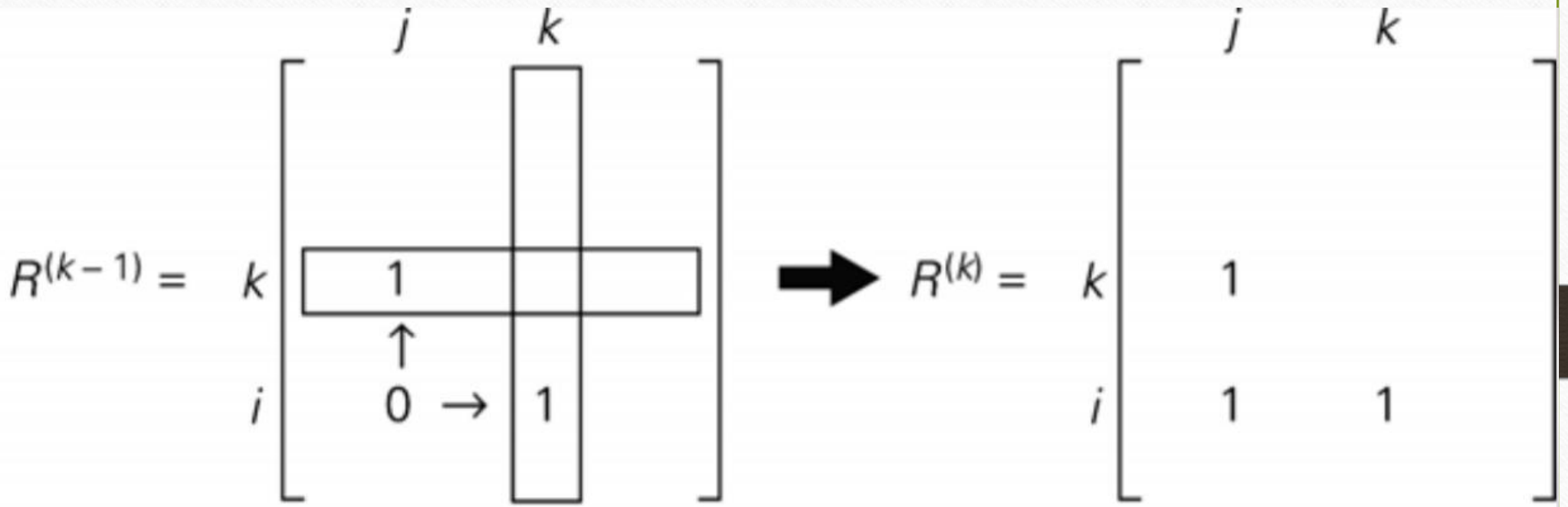
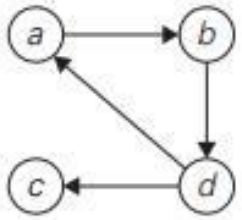


FIGURE 8.3 Rule for changing zeros in Warshall's algorithm



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

1's reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

1's reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

1's reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

1's reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

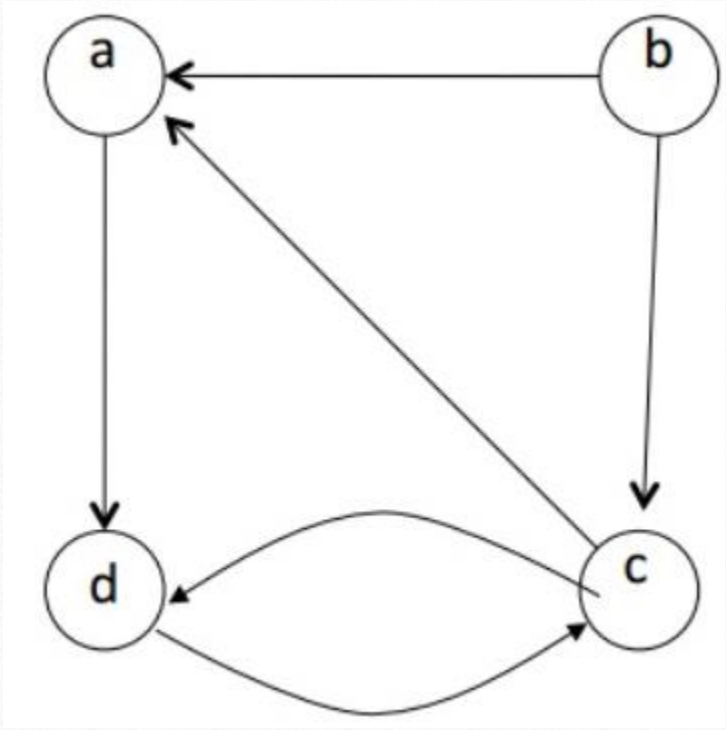
1's reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.13 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

Algorithm

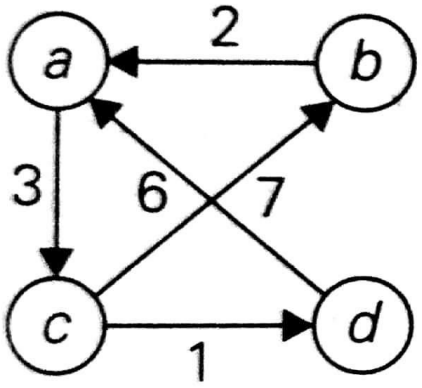
Warshall (A[1..n, 1..n])
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or ($R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j]$)
Return $R^{(n)}$

Todo



Floyd's Algorithm

- Is used to find **all-pairs shortest-paths** for a given weighted connected graph (directed or undirected, without cycle of negative length)
- Finds the length of the shortest paths from each vertex to every other vertices
- R. Floyd is the inventor of the algorithm



(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 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

- Let W be a n x n weight matrix to store the edge weights
- Let D be a n x n distance matrix which stores the shortest paths

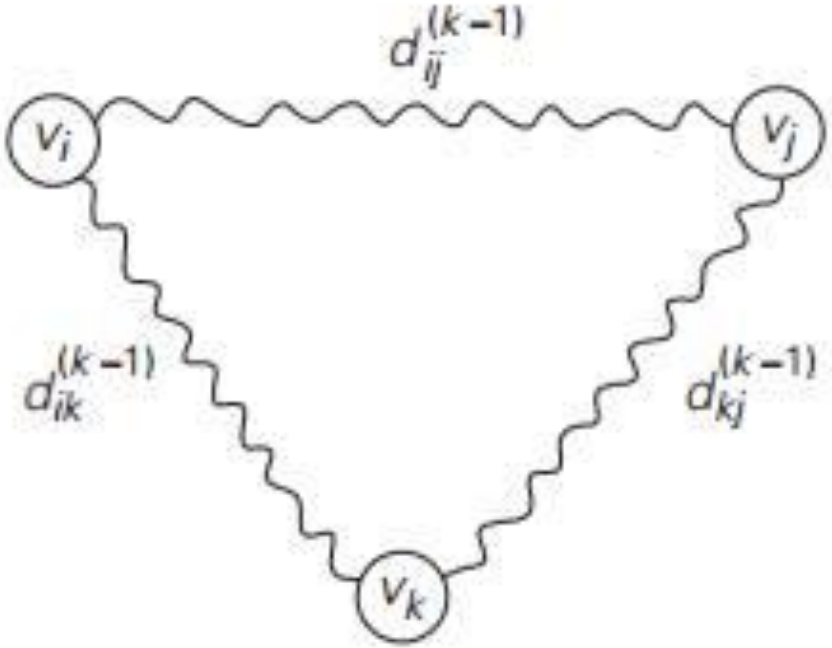
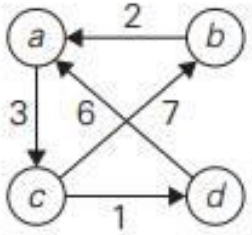


FIGURE 8.15 Underlying idea of Floyd's algorithm.

$$d_{ij}^{(k)} = \min \{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \text{ for } k>=1, d_{ij}^{(0)} = w_{ij}$$



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e., just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e., a and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e., a, *b*, and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e., a, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

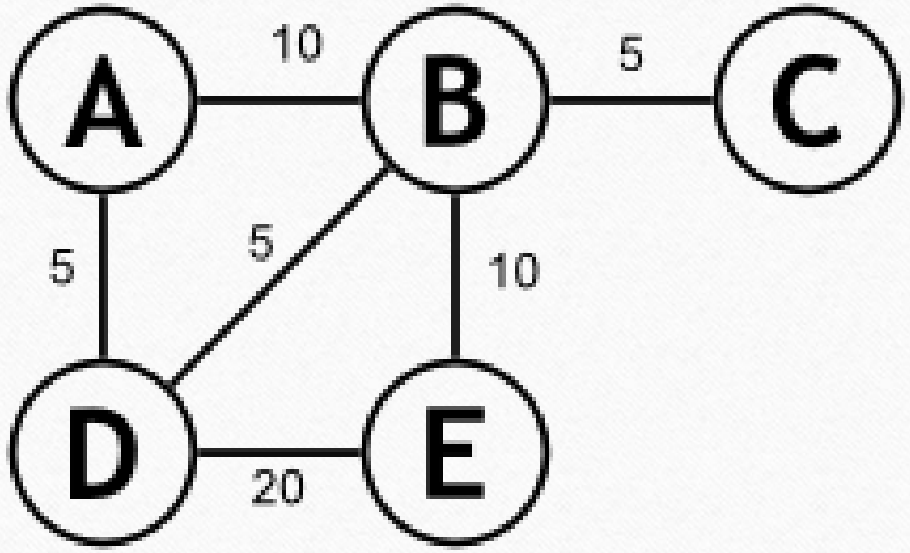
FIGURE 8.16 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

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 ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

- **Time efficiency is $\theta(n^3)$**

Todo



Optimal Binary Search Tree

- One of the principal applications of BST is to implement a dictionary, with operations to search, insert and delete
- If the probabilities of searching for elements of a set are known, then, is it possible to construct an Optimal BST so that the number of comparisons in a search is the smallest possible

- Consider 4 keys A, B, C and D with its search probabilities 0.1, 0.2, 0.4 and 0.3 respectively
- The average number of comparisons in a successful search for the two possible trees is given below

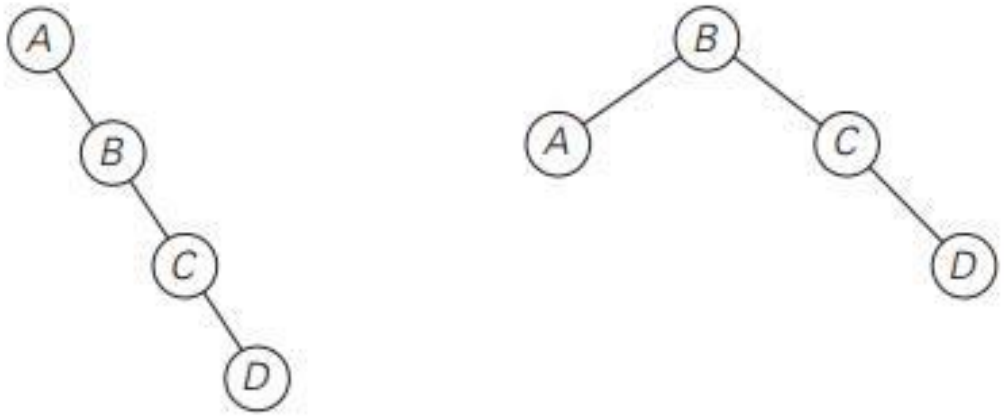


FIGURE 8.7 Two out of 14 possible binary search trees with keys *A*, *B*, *C*, and *D*.

- First tree - 0.1*1 + 0.2*2 + 0.4*3 + 0.3*4 = 2.9
- Second tree - 0.1*2 + 0.2*1 + 0.4*2 + 0.3*3 = 2.1 Is it optimal?

Method

- Let $a_1, a_2, \ldots a_n$ be distinct keys in ascending order
- Let $p_1, p_2, \ldots p_n$ be the search probabilities
- Let C[i, j] be the smallest average number of comparisons made in a successful search in a BST T_{ij}
- C[1, n] is the smallest average number of comparisons in a BST from a_1 to a_n

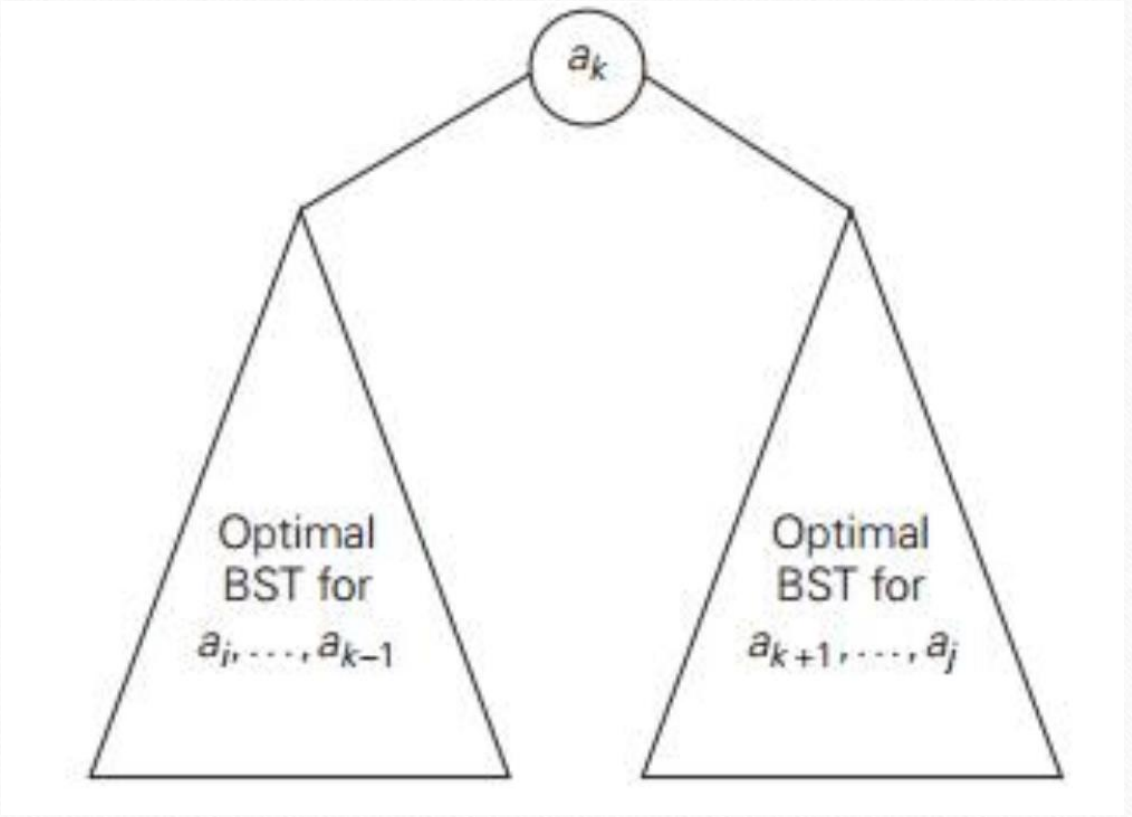


Fig 8.9: BST with root a_k and two optimal binary search subtrees T_i^{k-1} and T^j_{k+1}

$$C[i, j] = \min_{i \le k \le j} \{ C[i, k-1] + C[k+1, j] \} + \sum_{s=i}^{j} p_s \qquad \text{for } 1 \le i \le j \le n$$

$$C[i, i-1] = 0 \qquad \text{for } 1 \le i \le n+1$$

$$C[i, i] = p_i \qquad \text{for } 1 \le i \le n$$

Goal is to compute C[1, n]

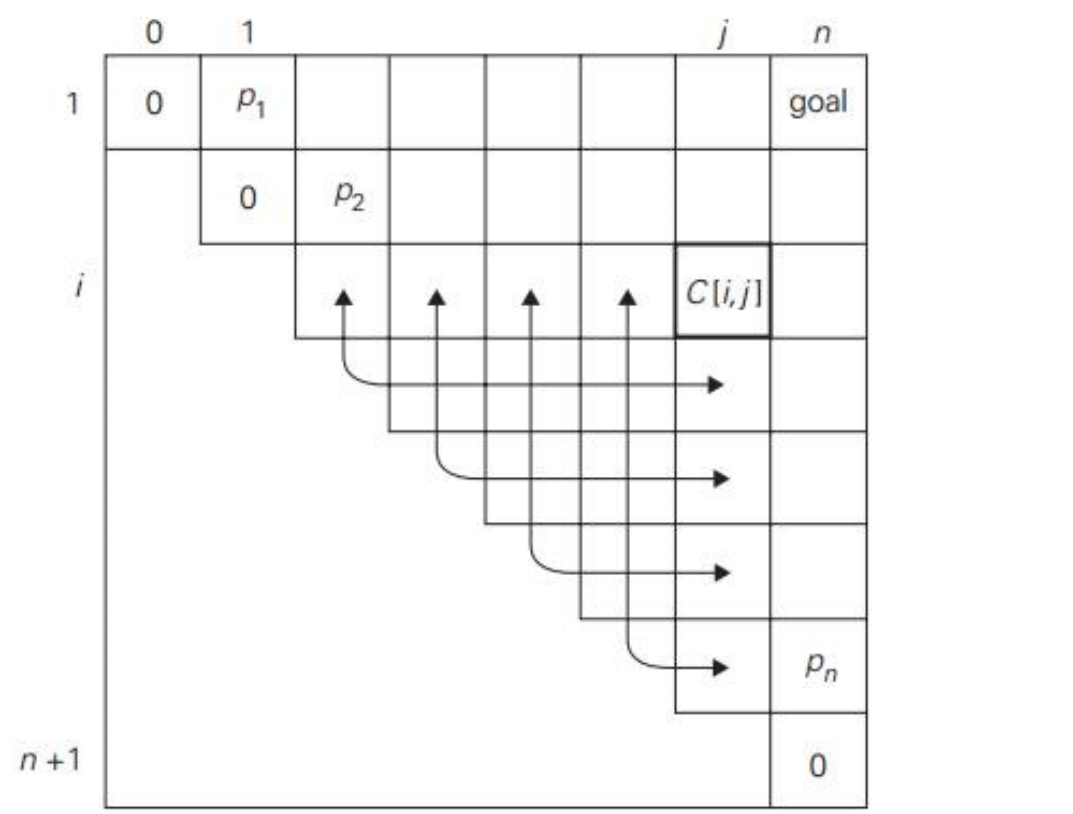


Table of the dynamic programming algorithm for constructing an optimal binary search tree.

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

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

Let us compute $C(1, 2)$:

$$C(1,2) = \min \left\{ \begin{array}{ll} k=1: & C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k=2: & C(1,1) + C(3,2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\}$$

$$= 0.4.$$

main table

	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

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

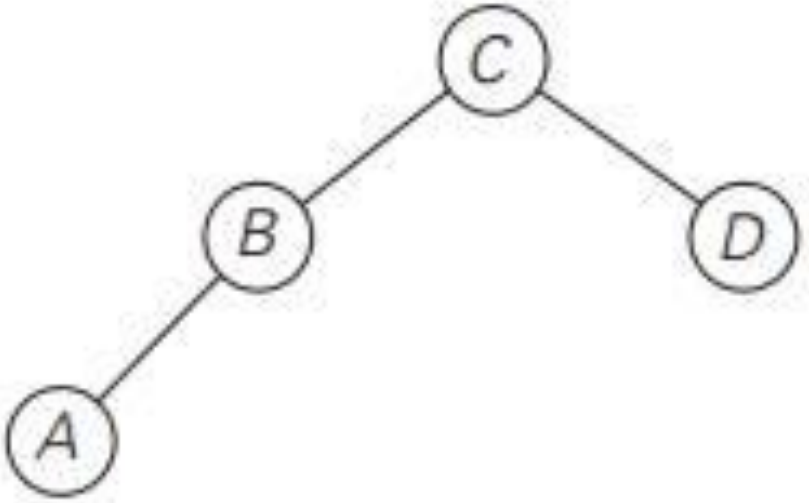


FIGURE 8.10 Optimal binary search tree for the example.

ALGORITHM *OptimalBST*$(P[1..n])$

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//        optimal BST and table R of subtrees' roots in the optimal BST
for i ← 1 to n do
    C[i, i − 1] ← 0
    C[i, i] ← P[i]
    R[i, i] ← i
C[n + 1, n] ← 0
for d ← 1 to n − 1 do //diagonal count
    for i ← 1 to n − d do
        j ← i + d
        minval ← ∞
        for k ← i to j do
            if C[i, k − 1] + C[k + 1, j] < minval
                minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
        R[i, j] ← kmin
        sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
        C[i, j] ← minval + sum
return C[1, n], R
```

Todo

Key	1	2	3	4	5
Probability	0.24	0.22	0.23	0.3	0.01

Knapsack Problem (0/1 Knapsack)

- Given n items of known weights w_1, …… w_n and values v_1 ……v_n and a knapsack of capacity W, find the most valuable subset of items that fit into the knapsack
- To design a dynamic programming algorithm a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances is needed

- Let V[i, j] be the value of an optimal solution to the instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j

- $V[i, j] = \begin{cases} \max\{ V[i-1, j], \quad vi + V[i-1, j-wi]\} & if\ j - wi \geq 0 \\ V[i-1, j] & if\ j - wi < 0 \end{cases}$

- V[0, j] = 0 for j ≥ 0

- V[i, 0] = 0 for i ≥ 0

- Goal is to find V[n, W], the maximal value of a subset of the n given items that fit into the knapsack of capacity W

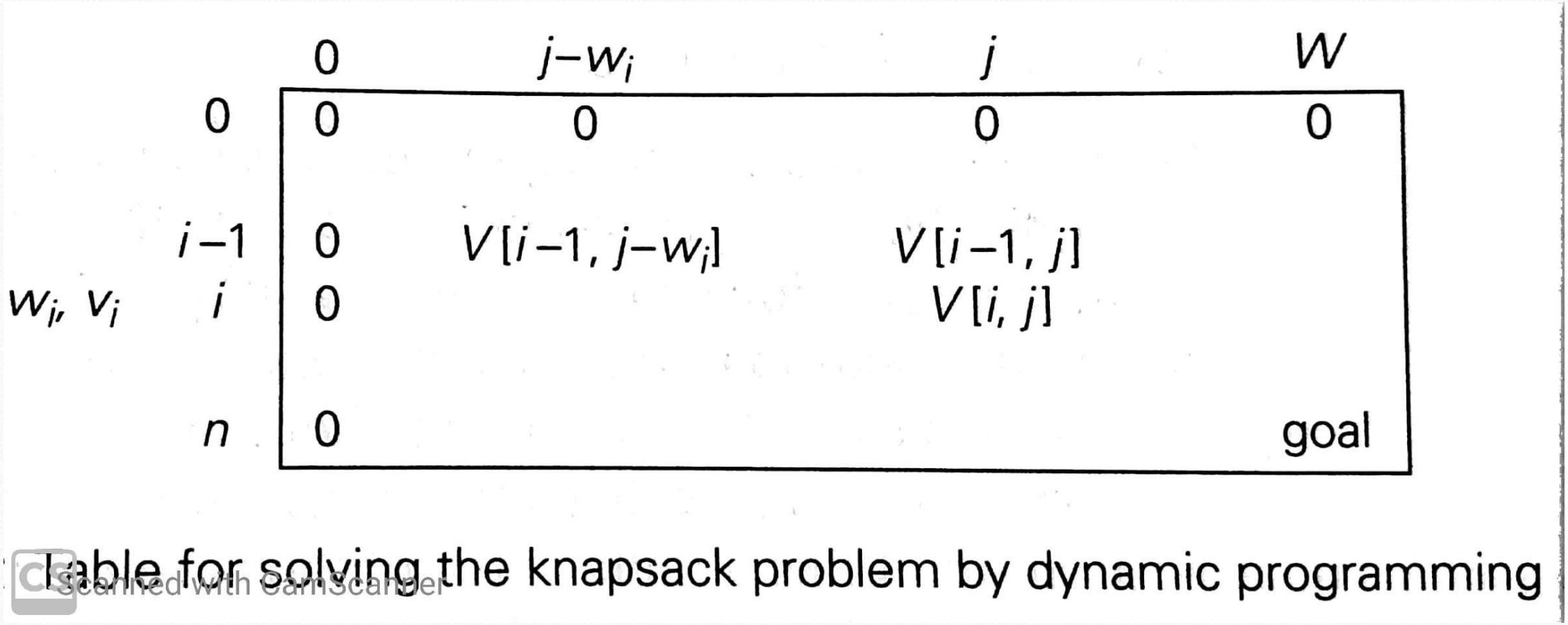


Table for solving the knapsack problem by dynamic programming

EXAMPLE 1 Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

- Thus maximal value is V[4, 5] = 37

- To **find the composition of an optimal subset**, trace back computations in the table entry
- V[4, 5] ≠ V[3, 5] Hence item 4 is included in the optimal subset with 5 - 2 = 3 remaining units of knapsack capacity
- V[3, 3] = V[2, 3] Hence item 3 is not part of optimal subset
- V[2, 3] ≠ V[1, 3] Hence item 2 is included in optimal subset with 3 – 1 = 2 remaining units of knapsack capacity
- V[1, 2] ≠ V[0, 2] Hence item 1 is included in optimal subset with 2 – 2 = 0 remaining units of knapsack capacity
- Therefore optimal subset contains {item 1, item 2, item 4}

Todo

item	weight	value
1	3	$25
2	2	$20
3	1	$15
4	4	$40
5	5	$50

capacity w=6

Memory Functions

- Dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping sub problems
- The direct top-down approach solves common sub problems more than once and hence is inefficient
- The classic dynamic programming approach works-bottom up, it fills a table to all smaller problems, but each of them is solved only once
- One notifiable point is that solutions to some of these sub problems are often not necessary for getting a solution for a given problem

- This drawback is not present in top-down approach and hence it is better to use the strengths of both top-down and bottom-up approaches
- The goal is to get a method that solves only sub problems which are necessary and does it only once
- This method is based on using **Memory Functions**
- This method uses top-down approach to solve the problem but in addition also maintains a table of the kind that is used in bottom-up approach
- Initially table values are filled with **null** symbol indicating that values are yet to be calculated
- Whenever a new value needs to be calculated, if the entry is not null, the value is retrieved, otherwise the value is calculated by the recursive call

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack's capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table V[0..n, 0..W] whose entries are initialized with -1's except for
//row 0 and column 0 initialized with 0's
if V[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    V[i, j] ← value
return V[i, j]
```

The function is called with i=n and j=W

EXAMPLE 1 Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$.

	i / j	0	1	2	3	4	5
	0	0	0	0	0	0	0
w_1= 2, v_1=12	1	0	-1	-1	-1	-1	-1
w_2= 1, v_2=10	2	0	-1	-1	-1	-1	-1
w_3= 3, v_3=20	3	0	-1	-1	-1	-1	-1
w_4= 2, v_4=15	4	0	-1	-1	-1	-1	-1

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

FIGURE 8.6 Example of solving an instance of the knapsack problem by the memory function algorithm.

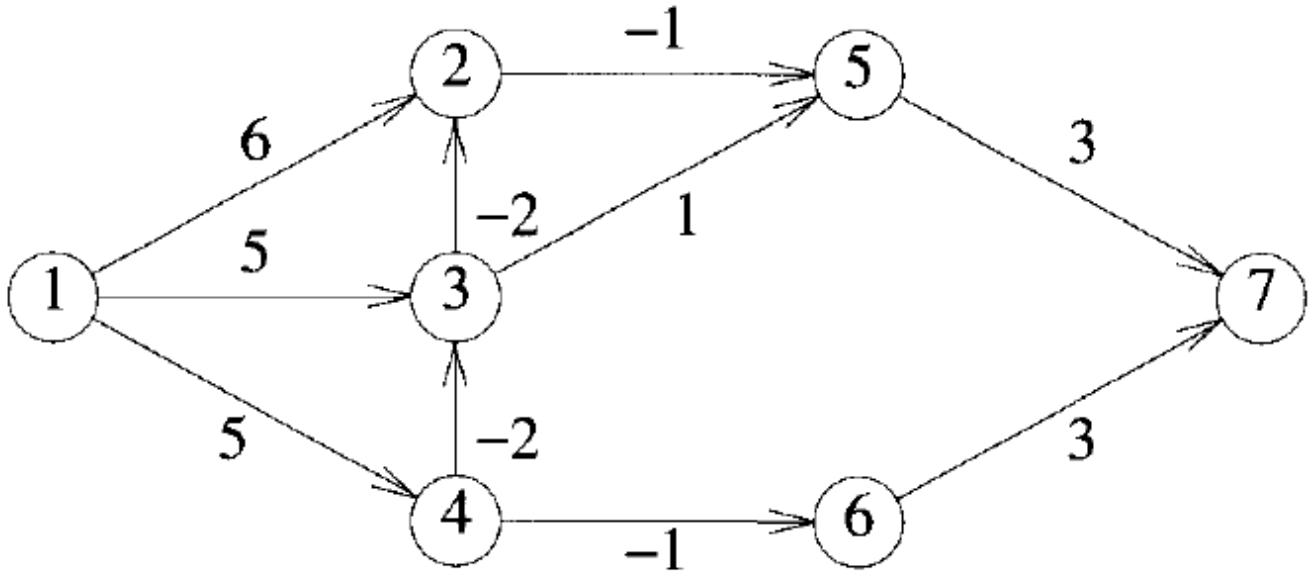
Todo

Item	1	2	3	4
Weight	2	6	7	3
Value	4	3	2	4
Knapsack capacity = 8				

Bellman Ford algorithm

- This algorithm is used to find single source shortest paths for a directed graph
- The graph may contain negative lengths, but not negative cycles
- The recurrence relation for calculating the distance *dist* is

$$\mathbf{Dist_k[u] = min\ \{\ dist_{k-1}[u],\ min_i\ \{\ dist_{k-1}[i] + cost[i, u]\}}\ \text{for } 2 \le k \le n-1$$



(a) A directed graph

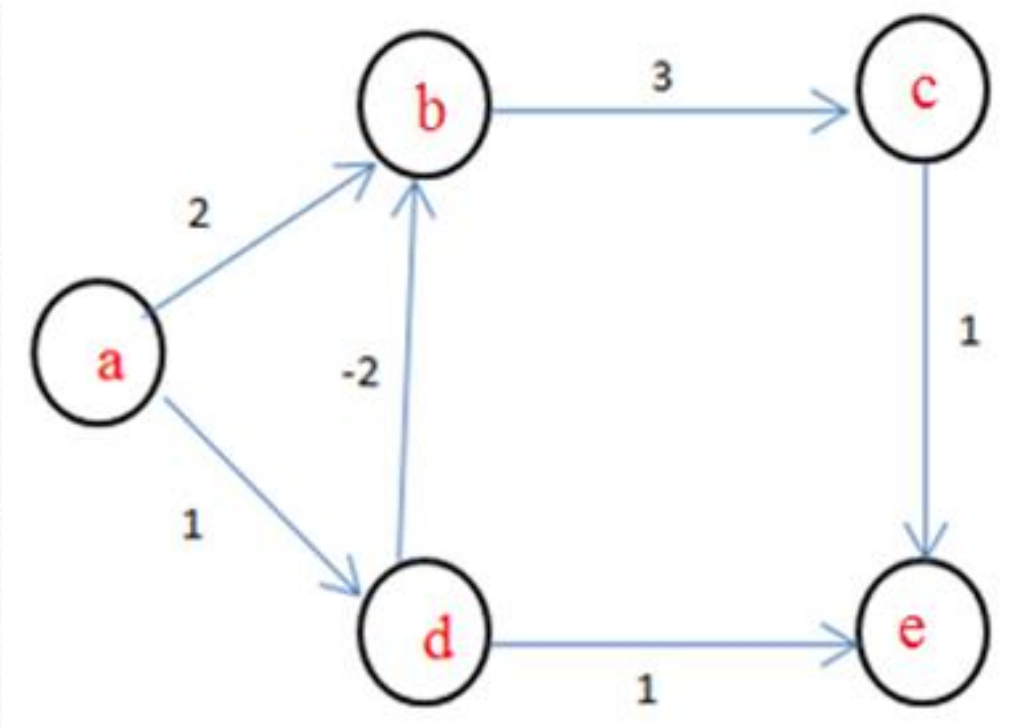
	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

$$dist^2[2] = \min\{dist^1[2], \min_i dist^1[i] + cost[i,2]\}$$
$$= \min\{6, 0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty, \infty+\infty\} = 3$$

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
     for i := 1 to n do // Initialize dist.
          dist[i] := cost[v, i];
     for k := 2 to n − 1 do
          for each u such that u ≠ v and u has
                    at least one incoming edge do
               for each ⟨i, u⟩ in the graph do
                    if dist[u] > dist[i] + cost[i, u] then
                         dist[u] := dist[i] + cost[i, u];
}
```

Todo



Travelling Salesperson Problem

- Let G=(V, E) be a directed graph with edge costs c_{ij}
- Let |V|=n, the number of vertices in the graph
- A tour in a graph is a directed simple cycle that includes every vertex in V
- The cost of the tour is the sum of the cost of the edges on the tour
- The travelling salesperson problem is to find a tour of minimum cost
- Applications: Salesperson, postal van, robot arm to tighten nuts on a machinery, etc.

- Every tour consists of an edge(l, k) for some k ∈ V - {1} and a path from vertex k to vertex 1
- The path from vertex k to vertex 1 goes through each vertex in V - {1, k} exactly once
- It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices V - {l, k}
- Let g(i, S) be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1
- The function g(l, V - {1}) is the length of an optimal salesperson tour

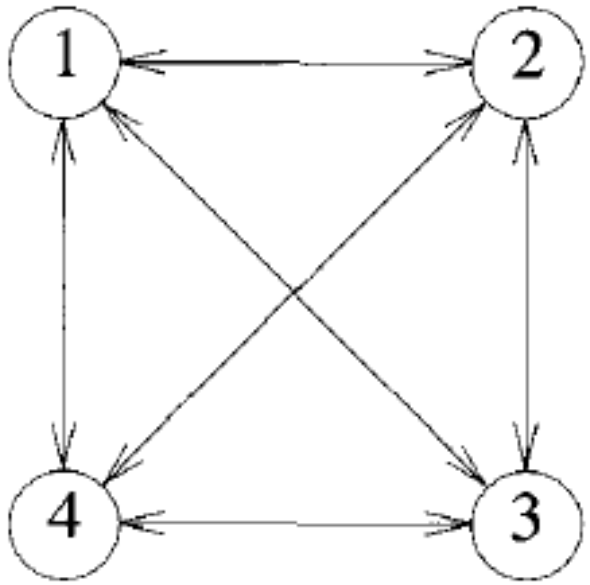
- From the principle of optimality it follows that

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\}$$

- Generalizing, we obtain

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\}$$

- When $|S|=0$, $g(i, \Phi) = c_{i1}$ $\quad 1 \le i \le n$
- Calculate $g(i, S)$ when $|S|=1$ and so on
- When $|S|<n-1$, $i \neq 1$, $1 \notin S$ and $i \notin S$



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Thus $g(2,\phi) = c_{21} = 5, g(3,\phi) = c_{31} = 6$, and $g(4,\phi) = c_{41} = 8$. Using (5.21), we obtain

$$\begin{array}{lllll} g(2,\{3\}) & = & c_{23} + g(3,\phi) & = & 15 \\ g(3,\{2\}) & = & 18 & & \\ g(4,\{2\}) & = & 13 & & \end{array} \qquad \begin{array}{lll} g(2,\{4\}) & = & 18 \\ g(3,\{4\}) & = & 20 \\ g(4,\{3\}) & = & 15 \end{array}$$

Next, we compute $g(i,S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{lcll} g(2,\{3,4\}) & = & \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} & = 25 \\ g(3,\{2,4\}) & = & \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} & = 25 \\ g(4,\{2,3\}) & = & \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} & = 23 \end{array}$$

Finally, from (5.20) we obtain

$$\begin{aligned} g(1,\{2,3,4\}) &= \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\ &= \min\{35, 40, 43\} \\ &= 35 \end{aligned}$$

Let J(i, S) be the value that gives the minimum value

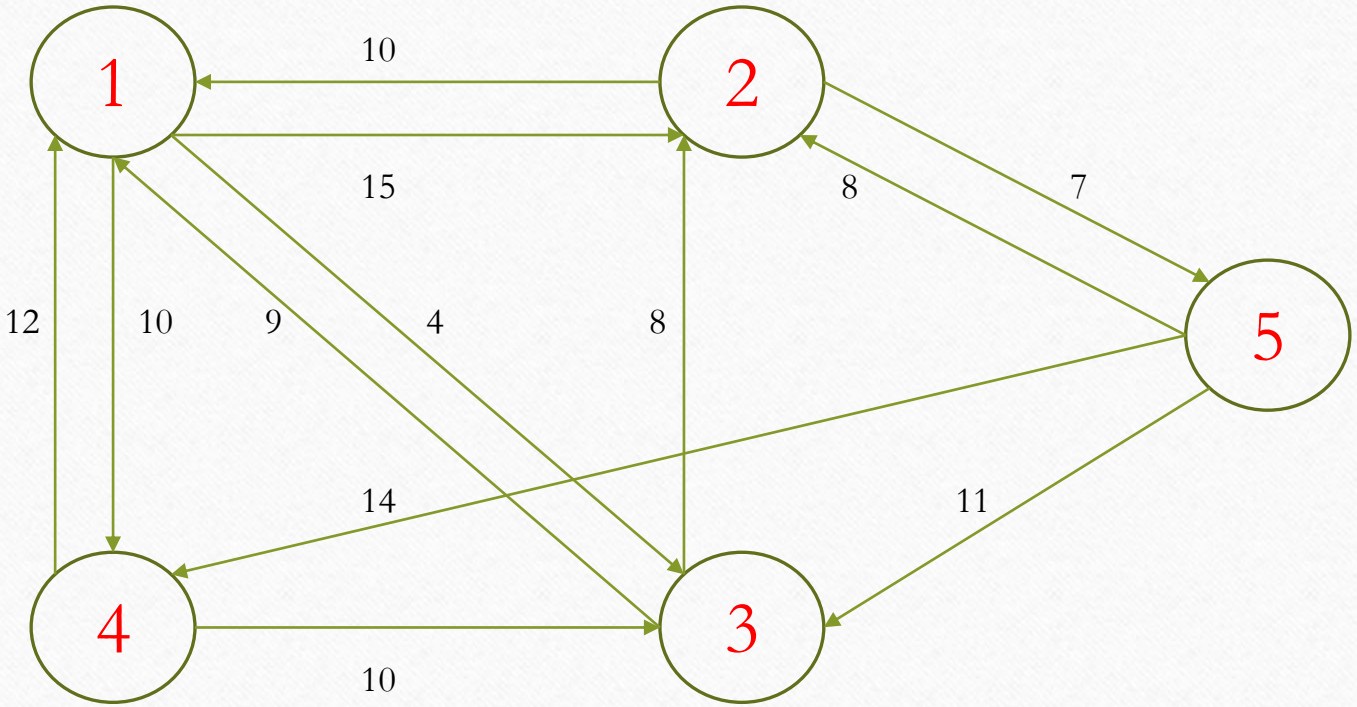
$J(1, \{2, 3, 4\}) = 2$

$J(2, \{3, 4\}) = 4$

$J(4, \{3\}) = 3$

Hence the optimal tour is 1, 2, 4, 3, 1

Todo



Reliability Design

- The problem is design a reliable system within the given cost
- Let r_i be the reliability of the device D_i
- The reliability of the entire system where devices are connected in parallel is Πr_i
- For example n=4, r_i=0.9 $1 \leq n \leq 4$. $\Pi r_i = 0.6561$
- Hence devices need to be duplicated to increase reliability
- Multiple copies of the same device are connected in parallel

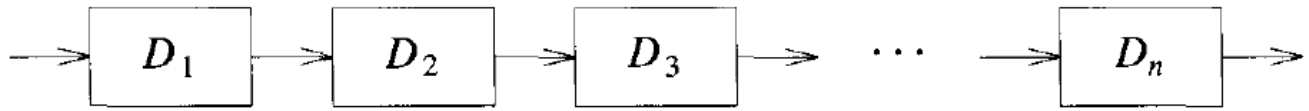


Figure 5.19 n devices D_i, $1 \leq i \leq n$, connected in series

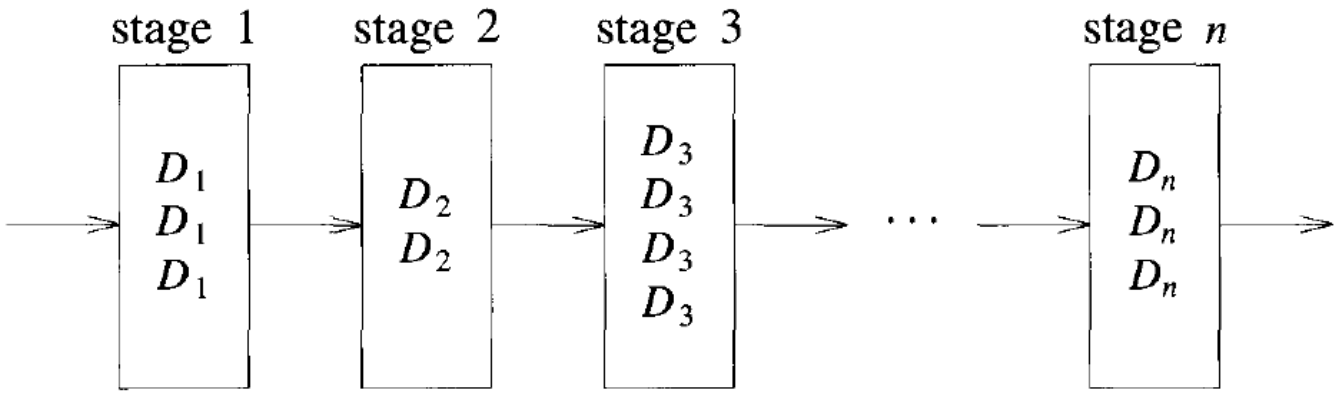


Figure 5.20 Multiple devices connected in parallel in each stage

- If stage i contains mi copies of device D_i, then the probability that all m_i malfunctions is $(1-r_i)^{mi}$
- Hence the reliability of stage i is $1- (1-r_i)^{mi}$
- if $r_i = 0.99$ and $m_i = 2$, then the reliability of stage i is $1-(1-0.99)^2 = 0.9999$
- Let the reliability of the stage i is $\Phi_i(m_i)$
- Then the reliability of the system is $\Pi_{1\leq i\leq n}\Phi_i(m_i)$

- The problem is to maximize reliability of the system under a cost constraint
- Let c_i be the cost of each unit of device i and c be the maximum allowable cost of the system being designed
- Therefore the following maximization problem has to be solved

$$\text{maximize } \Pi_{1 \le i \le n} \Phi_i(m_i)$$

$$\text{subject to } \sum_{i=1}^{n} c_i m_i \le c$$

$$m_i \ge 1 \text{ and integer, } 1 \le i \le n$$

- Each mi must be in the range $1 \le m_i \le u_i$ where

$$u_i = \lfloor (c - \sum_{i=1}^{n} c_i)/ci \rfloor$$

D_1, D_2, D_3 have a cost of 30, 15, 20 and reliability 0.9, 0.8, 0.5 respectively. The cost constraint is 105

Device	Cost c_i	Reliability r_i
D_1	30	0.9
D_2	15	0.8
D_3	20	0.5

Maximum reliability of the system = **0.648**

Total Cost = **100**

No. of copies of $D_1 = m_1 = \mathbf{1}$

$D_2 = m_2 = \mathbf{2}$

$D_3 = m_3 = \mathbf{2}$

Todo

D_1, D_2, D_3, D_4 have a cost of 25, 40, 30, 50 and reliability 0.8, 0.7, 0.6, 0.9 respectively. The cost constraint is 225. Find the Optimal reliability of the system and also the cost.