Design and Analysis of Algorithms

Module 4: Dynamic Programming

Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- Transitive Closure:
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- Reliability design

Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- Transitive Closure:
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- Reliability design

Dynamic Programming

- Dynamic programming is a technique for solving problems with **overlapping subproblems**.
  - subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems.
  - DP suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.
- The Dynamic programming can be used when the solution to a problem can be viewed as the result of **sequence of decisions**.

Example-1: Knapsack Problem

Formal description: Given two n-tuples of positive numbers

$$\langle v_1, v_2, \ldots, v_n \rangle \quad \text{and} \quad \langle w_1, w_2 \ldots, w_n \rangle,$$

and $W > 0$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of files to store) that

$$\text{maximizes} \quad \sum_{i \in T} v_i,$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Example-2: File merging problem

Example 4.9 The files x_1, x_2, and x_3 are three sorted files of length $30, 20,$ and 10 records each. Merging x_1 and x_2 requires 50 record moves. Merging the result with x_3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If, instead, we first merge x_2 and x_3 (taking 30 moves) and then x_1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first. □

Principle of Optimality

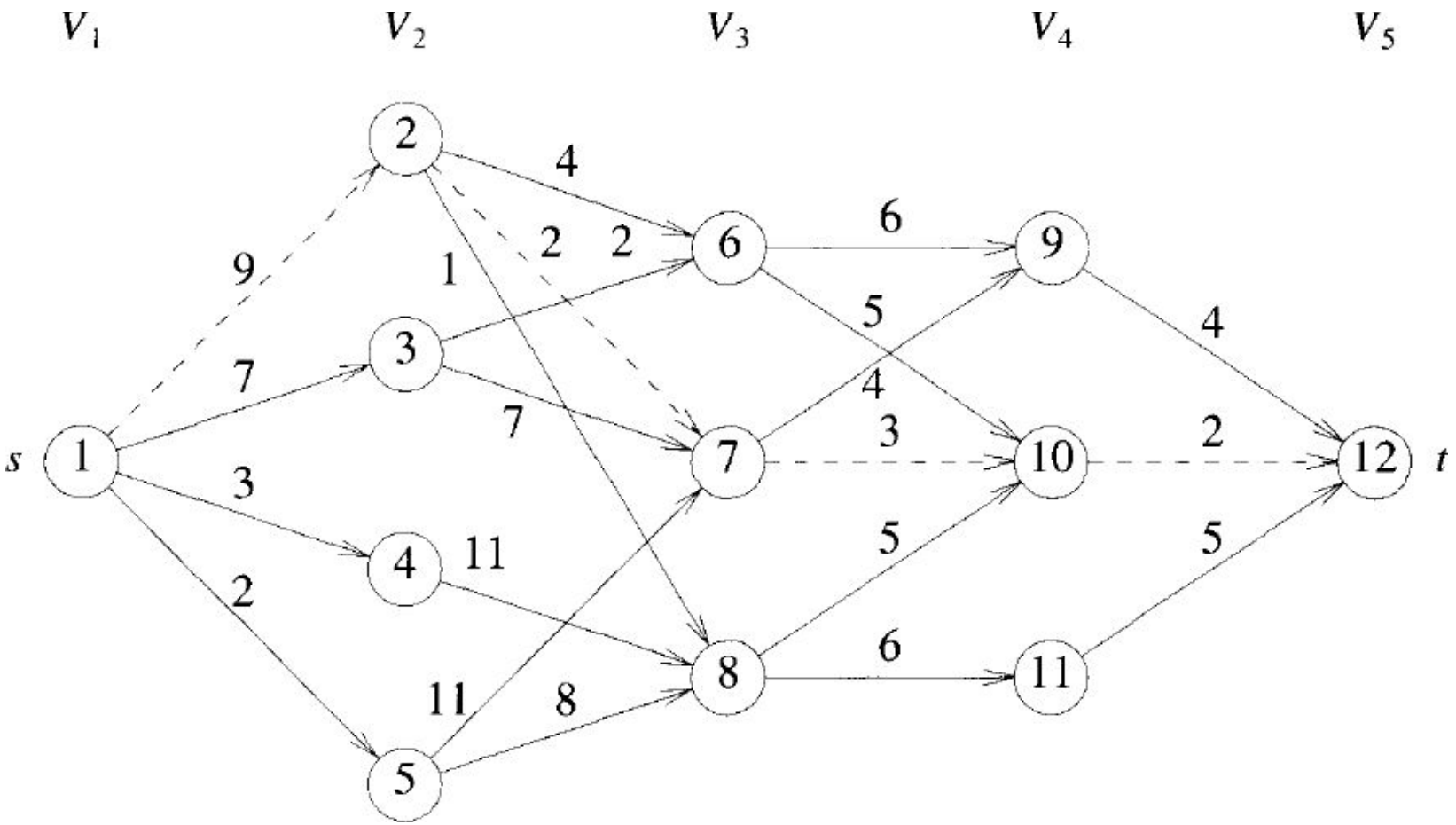
Definition 5.1 [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. □

Difference between greedy and DP?

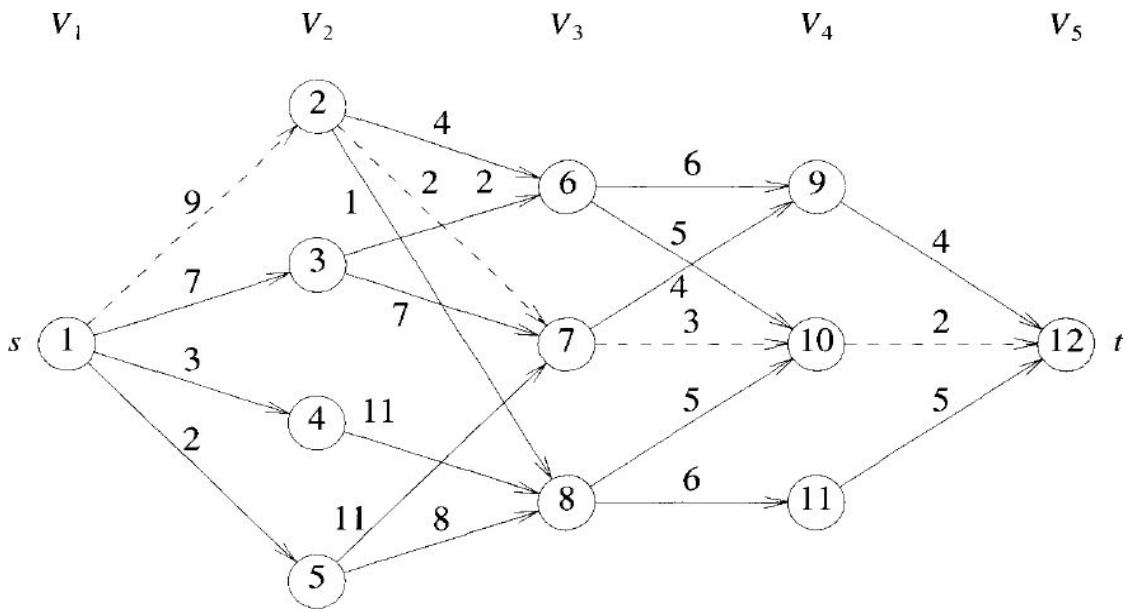
Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - **Multistage Graphs**
- Transitive Closure:
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- Reliability design

Multistage Graphs



DP Solution using forward approach



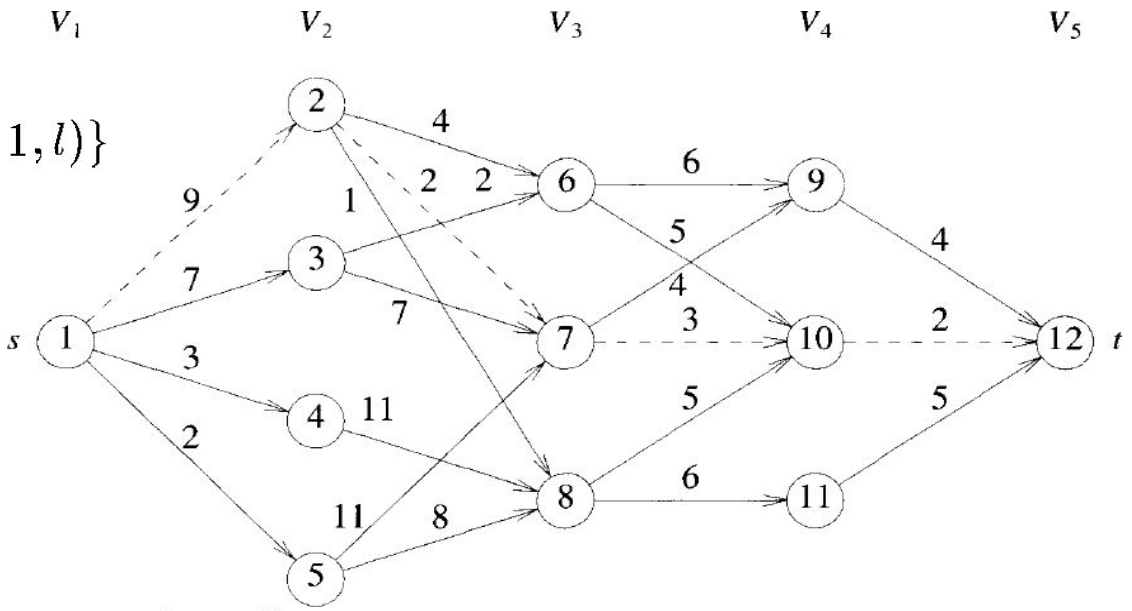
Cost of node **j** at stage **i** to reach terminal node **t**

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

Stage number

Node

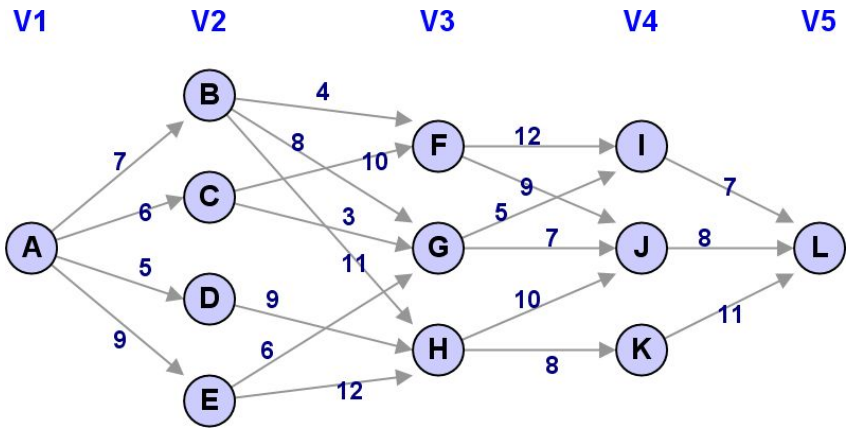
$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\}$$



cost(5,12)=0

cost(4,9)=c(9,12) = 4
cost(4,10)=c(10,12)= 2
cost(4,11)=c(11,12)= 5

$$\begin{aligned}
cost(3,6) &= \min\{6 + cost(4,9), 5 + cost(4,10)\} \\
&= 7 \\
cost(3,7) &= \min\{4 + cost(4,9), 3 + cost(4,10)\} \\
&= 5 \\
cost(3,8) &= 7 \\
cost(2,2) &= \min\{4 + cost(3,6), 2 + cost(3,7), 1 + cost(3,8)\} \\
&= 7 \\
cost(2,3) &= 9 \\
cost(2,4) &= 18 \\
cost(2,5) &= 15 \\
cost(1,1) &= \min\{9 + cost(2,2), 7 + cost(2,3), 3 + cost(2,4), \\
&\qquad 2 + cost(2,5)\} \\
&= 16
\end{aligned}$$



cost(4,I) = c(I,L) = 7
cost(4,J) = c(J,L) = 8
cost(4,K) = c(K,L) = 11

cost(3,F) = min { c(F,I) + cost(4,I) | c(F,J) + cost(4,J) }
cost(3,F) = min { 12 + 7 | 9 + 8 } = 17
cost(3,G) = min { c(G,I) + cost(4,I) | c(G,J) + cost(4,J) }
cost(3,G) = min { 5 + 7 | 7 + 8 } = 12
cost(3,H) = min { c(H,J) + cost(4,J) | c(H,K) + cost(4,K) }
cost(3,H) = min { 10 + 8 | 8 + 11 } = 18
cost(2,B) = min { c(B,F) + cost(3,F) | c(B,G) + cost(3,G) | c(B,H) + cost(3,H) }
cost(2,B) = min { 4 + 17 | 8 + 12 | 11 + 18 } = 20
cost(2,C) = min { c(C,F) + cost(3,F) | c(C,G) + cost(3,G) }
cost(2,C) = min { 10 + 17 | 3 + 12 } = 15
cost(2,D) = min { c(D,H) + cost(3,H) }
cost(2,D) = min { 9 + 18 }= 27
cost(2,E) = min { c(E,G) + cost(3,G) | c(E,H) + cost(3,H) }
cost(2,E) = min { 6 + 12 | 12 + 18 } = 18
cost(1,A) = min { c(A,B) + cost(2,B) | c(A,C) + cost(2,C) | c(A,D) + cost(2,D) | c(A,E) + cost(2,E) }
cost(1,A) = min { 7 + 20 | 6 + 15 | 5 + 27 | 9 + 18 } = 21

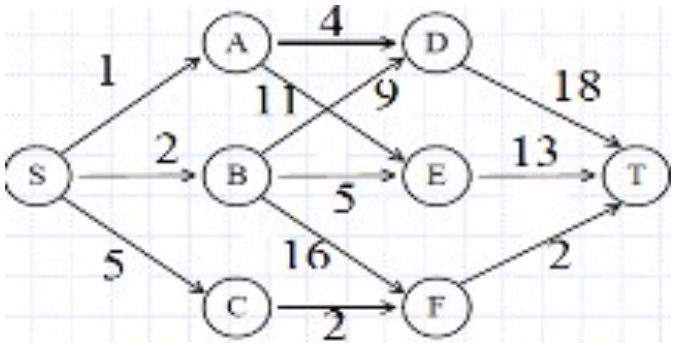
Optimal Path:
A-C-G-I-L

Reused costs are colored!

Principle of optimality w.r.t. Multistage graph

- Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions $D_1, D_2, \ldots, D_n$. If this sequence is optimal, then the last k decisions, $1 < k < n$ must be optimal.
- e.g. the shortest path problem

  If $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i_1 to j



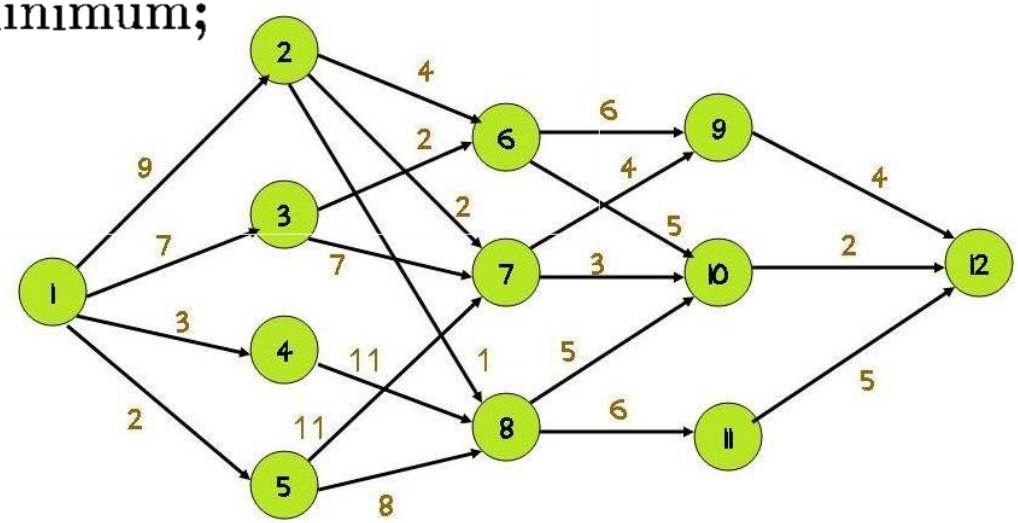
cost(3,D) = c(D,T) = 18
cost(3,E) = c(E,T) = 13
cost(3,F) = c(F,T) = 2

cost(2,A) = min { c(A,D) + cost(3,D) | c(A,E) + cost(3,E) }
cost(2,A) = min { 4 + 18 | 11 + 13 } = 22
cost(2,B) = min { c(B,D) + cost(3,D) | c(B,E) + cost(3,E) | c(B,F) + cost(3,F) }
cost(2,B) = min { 9 + 18 | 5 + 13 | 16+2 } = 18
cost(2,C) = min { c(C,F) + cost(3,F) }
cost(2,C) = min { 2 + 2 } = 4
cost(1,S) = min { c(S,A) + cost(2,A) | c(S,B) + cost(2,B) | c(S,C) + cost(2,C) }
cost(1,S) = min { 1 + 22 | 2 + 18 | 5 + 4 } = 9

Optimal Path: S-C-F-L

```
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]];
}
```

```
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;
}
```



When j=11, only one candidate for r, r=12 When j=10, only one candidate for r, r=12 When j=9, only one candidate for r, r=12

cost(11) = c(11,12)+cost(12)=5+0=**5**; d(11)=12

cost(10) = c(10,12)+cost(12)=2+0=**2**; d(10)=12

cost(9) = c(9,12)+cost(12)=4+0=**4**; d(9)=12

When j=8, two candidates for r, r=10,11

cost(8) = min { c(8,10)+cost(10)=5+2=7; c(8,11)+cost(11)=6+5=11; } = **7** d(8)=10

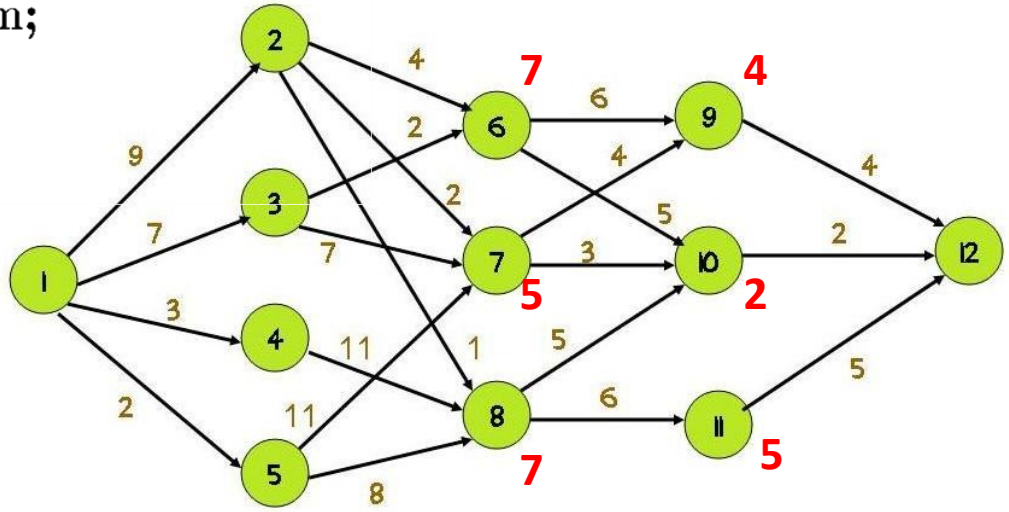
When j=7, two candidates for r, r=9,10

cost(7) = min { c(7,9)+cost(9)=4+4=8; c(7,10)+cost(10)=3+2=5; } = **5** d(7)=10

When j=6, two candidates for r, r=9,10

cost(6) = min { c(6,9)+cost(9)=6+4=10; c(6,10)+cost(10)=5+2=7; } = **7** d(6)=10

```
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;
}
```



When j=11, cost(11) =**5**; d(11)=12
When j=10, cost(10) =**2**; d(10)=12
When j=9, cost(9) =**4;** d(9)=12
When j=8, cost(8) = **7** d(8)=10
When j=7, cost(7) = **5** d(7)=10
When j=6, cost(6) = **7** d(6)=10

When j=5, two candidates for r, r=7,8 Cost(5) = d(5)=
When j=4, r=11 Cost(4) = d(4)=

When j=3, two candidates for r, r=6,7
`Cost(3) = d(3)=

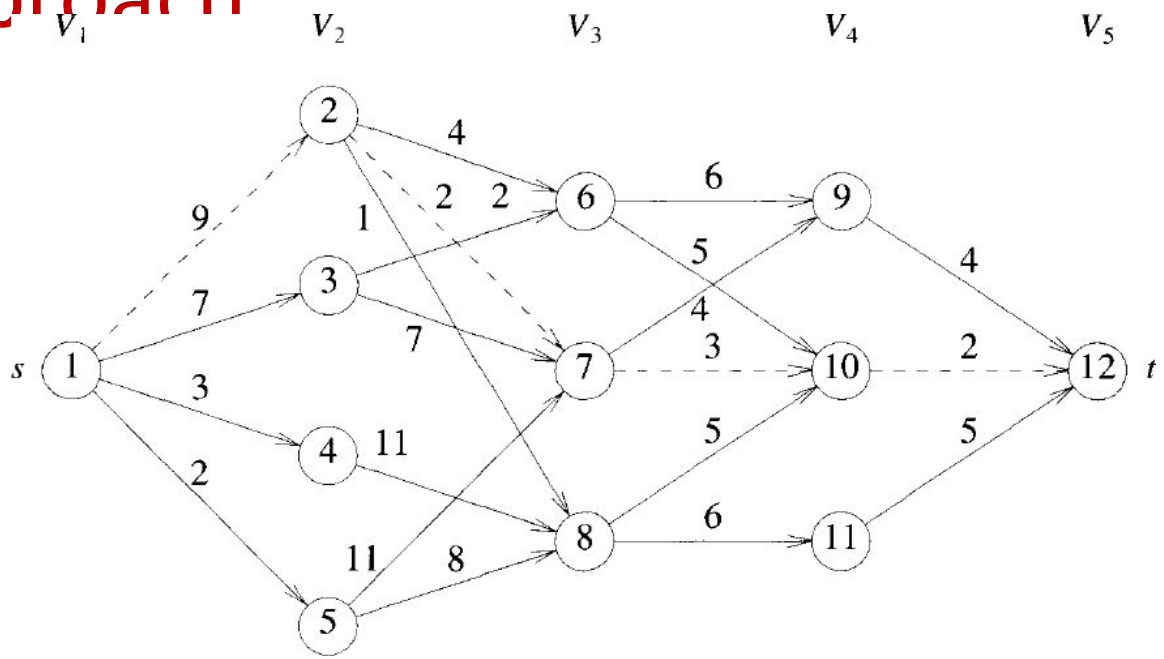
When j=2, three candidates for r, r=6,7,8 Cost(2) = d(2)=

Analysis

The complexity analysis of the function FGraph is fairly straightforward. If G is represented by its adjacency lists, then r in line 9 of Algorithm 5.1 can be found in time proportional to the degree of vertex j. Hence, if G has $|E|$ edges, then the time for the **for** loop of line 7 is $\Theta(|V| + |E|)$. The time for the **for** loop of line 16 is $\Theta(k)$. Hence, the total time is $\Theta(|V| + |E|)$. In addition to the space needed for the input, space is needed for $cost[\ ]$, $d[\ ]$, and $p[\ ]$.

Θ(|V| + |E|)

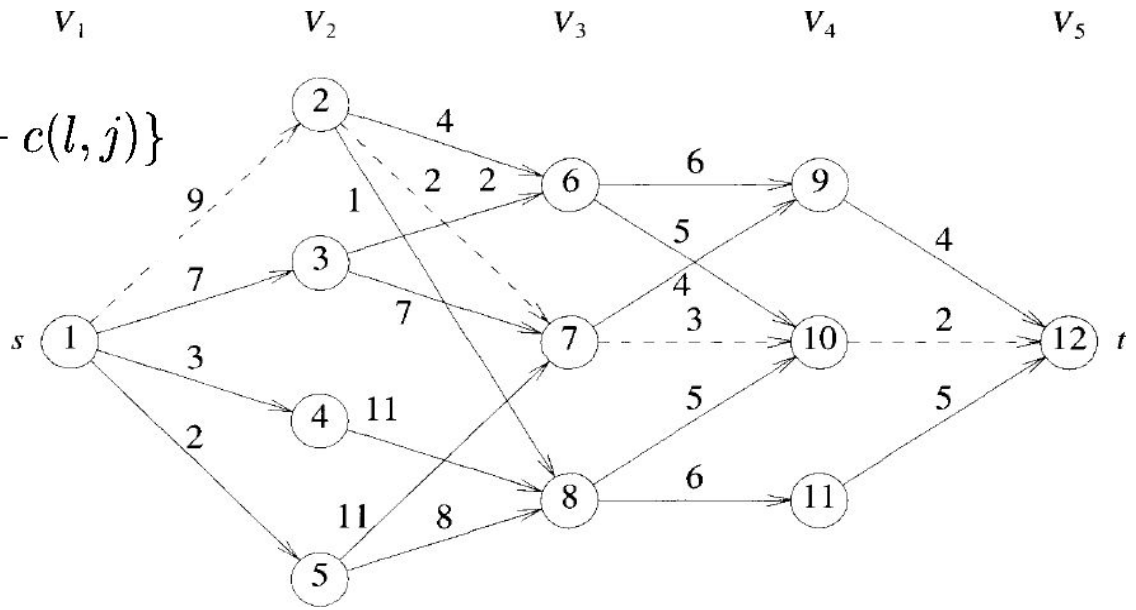
Backward Approach



Cost of node **j** at stage **i** to from source node **s**

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

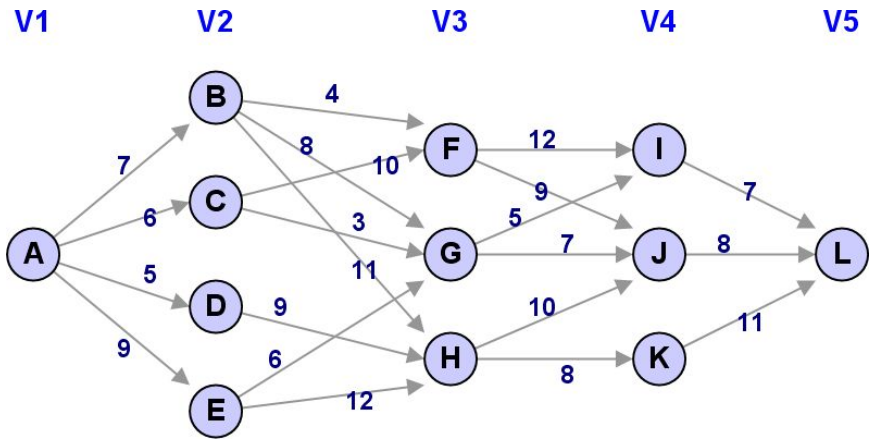
$$bcost(i,j) = \min_{\substack{l \in V_{i-1} \\ \langle l,j \rangle \in E}} \{bcost(i-1,l) + c(l,j)\}$$



bcost(2,2) = 9
bcost(2,3)=7
bcost(2,4)=3
bcost(2,5)=2

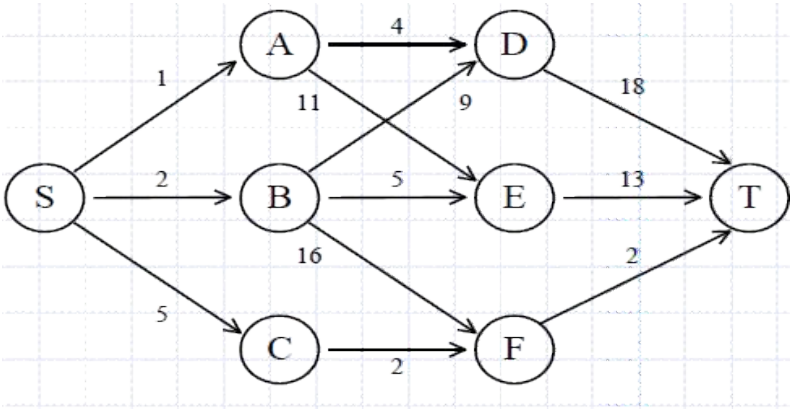
$$\begin{aligned}
bcost(3,6) &= \min\{bcost(2,2) + c(2,6), bcost(2,3) + c(3,6)\} \\
&= \min\{9+4, 7+2\} \\
&= 9 \\
bcost(3,7) &= 11 \\
bcost(3,8) &= 10 \\
bcost(4,9) &= 15 \\
bcost(4,10) &= 14 \\
bcost(4,11) &= 16 \\
bcost(5,12) &= 16
\end{aligned}$$

Example




```
bcost(2,B) = c(A,B) = 7
bcost(2,C) = c(A,C) = 6
bcost(2,D) = c(A,D) = 5
bcost(2,E) = c(A,E) = 9.
bcost(3,F) = min { c(B,F) + bcost(2,B) | c(C,F) + bcost(2,C) }
bcost(3,F) = min { 4 + 7 | 10 + 6 } =   11
bcost(3,G) = min { c(B,G) + bcost(2,B) | c(C,G) + bcost(2,C) | c(E,G)
   + bcost(2,E) }
bcost(3,G) = min { 8 + 7 | 3 + 6 | 6 + 9 } =   9
bcost(3,H) = min { c(B,H) + bcost(2,B) | c(D,H) + bcost(2,D) | c(E,H)
   + bcost(2,E) }
bcost(3,H) = min { 11 + 7 | 9 + 5 | 12 + 9 }    =   14
bcost(4,I) = min { c(F,I) + bcost(3,F) | c(G,I) + bcost(3,G) }
bcost(4,I) = min { 12 + 11 | 5 + 9 }    =   14
bcost(4,J) = min { c(F,J) + bcost(3,F) | c(G,J) + bcost(3,G) | c(H,J)
   + bcost(3,H) }
bcost(4,J) = min { 9 + 11 | 7 + 9 | 10 + 14 }   =   16
bcost(4,K) = min { c(H,K) + cost(3,H) }
bcost(4,K) = min { c(I,L) + bcost(4,I) | c(J,L)  + bcost(4,J) | c(K,L)
   + bcost(4,K) }
bcost(5,L) = min { 7 + 14 | 8 + 16 | 11 + 22 }  =
21
Path: A-C-G-I-L
cost:21
```

Reused costs are colored!



Algorithm BGraph(G, k, n, p)
// Same function as FGraph
{
    $bcost[1] := 0.0$;
    for $j := 2$ **to** n **do**
    { // Compute $bcost[j]$.
        Let r be such that $\langle r, j \rangle$ is an edge of
        G and $bcost[r] + c[r, j]$ is minimum;
        $bcost[j] := bcost[r] + c[r, j]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := k - 1$ **to** 2 **do** $p[j] := d[p[j + 1]]$;
}

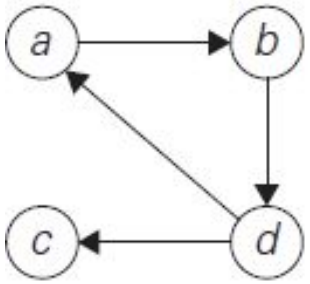
Multistage Graph

- Forward approach and backward approach:
  - Note that if the recurrence relations are formulated using the forward approach then the relations are solved backwards . i.e., beginning with the last decision
  - On the other hand if the relations are formulated using the backward approach, they are solved forwards.
- To solve a problem by using dynamic programming:
  - Find out the recurrence relations.
  - Represent the problem by a multistage graph.

Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- **Transitive Closure:**
  - **Warshall's Algorithm**,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- Reliability design

Transitive Closure



(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)

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

Definition: 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 element in the i^{th} row `and the j^{th} column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the i^{th} vertex to the j^{th} vertex; otherwise, t^{ij} is 0.

Transitive Closure

- We can generate the transitive closure of a digraph with the help of depth-first search or breadth-first search.
- Since this method traverses the same digraph several times, we can use a better algorithm called **Warshall's algorithm**.
- Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}.$$

Warshalls Algorithm

- The element in the i^{th} row and j^{th} column of matrix $R^{(k)}$ is equal to 1 if and only if
  - there exists a directed path of a positive length from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than k.

- This means that there exists a path from the i^{th} vertex v_i to the j^{th} vertex v^j with each intermediate vertex numbered not higher than k:

v_i, a list of intermediate vertices each numbered not higher than k, v_j

Warshalls Algorithm

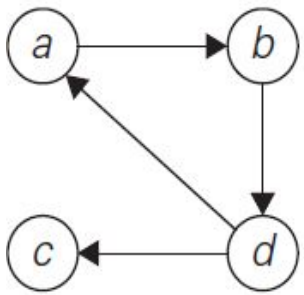
Two situations regarding this path are possible.

1. In the first, the list of its intermediate vertices **does not** contain the k^{th} vertex.

$$r_{ij}^{(k)} = r_{ij}^{(k-1)}$$

2. The second possibility is that path **does contain** the k^{th} vertex v_k among the intermediate vertices.

$$\left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right)$$



$R^{(0)} =$

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

$R^{(1)} =$

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

$R^{(2)} =$

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

$R^{(3)} =$

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

$R^{(4)} =$

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

Algorithm

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 $R^{(n)}$

Analysis

- Its time efficiency is **$\Theta(n^3)$.**
- We can make the algorithm to run faster by treating matrix rows as bit strings and employ the bitwise or operation available in most modern computer languages.
- Space efficiency
  - Although separate matrices for recording intermediate results of the algorithm are used, that can be avoided.

Outline

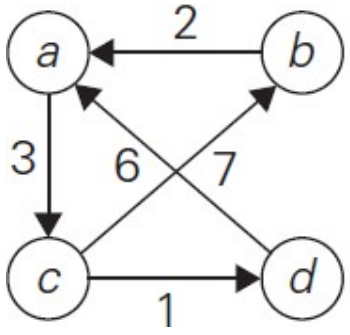
- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- Transitive Closure:
  - Warshall's Algorithm,
- **All Pairs Shortest Paths**:
  - **Floyd's Algorithm,**
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- Reliability design

All Pairs Shortest Paths

- **Problem definition:** Given a weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances—i.e., the lengths of the shortest paths - from each vertex to all other vertices.

- Applications:
  - Communications, transportation networks, and operations research.
  - pre-computing distances for motion planning in computer games.

All Pairs Shortest Paths



(a) Digraph.

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

(b) Its weight matrix.

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

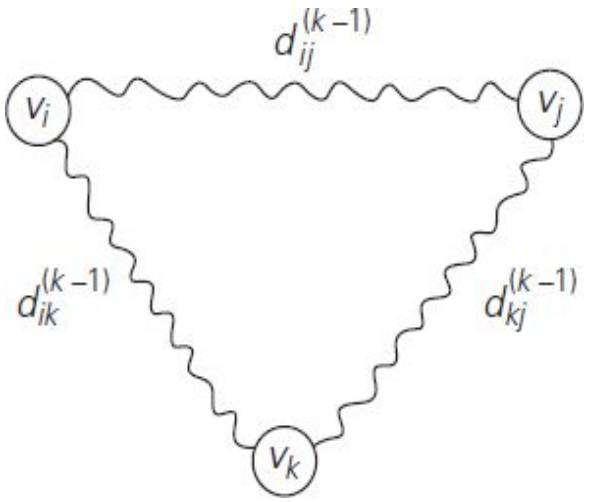
(c) Its distance matrix

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm.

It is called **Floyd's algorithm.**

Floyds Algorithm

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}.$$



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

Floyds Algorithm

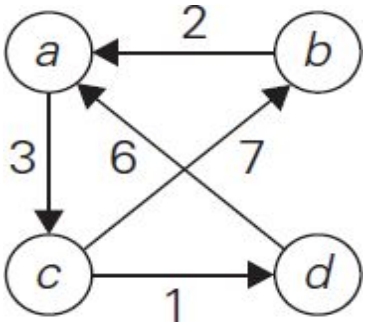
ALGORITHM *Floyd*$(W[1..n, 1..n])$

//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

Analysis

- Its time efficiency is $\Theta(n^3)$,
  - similar to the warshall's algorithm.

Example



$D^{(0)} =$

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

$D^{(1)} =$

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

$D^{(2)} =$

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

$D^{(3)} =$

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

$D^{(4)} =$

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

Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- Transitive Closure:
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,

- **Optimal Binary Search Trees**
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- Reliability design

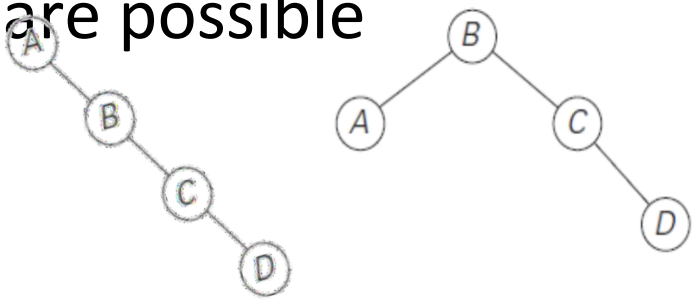
Optimal Binary Search Tree

- A binary search tree is one of the most important data structures in computer science.
- One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.
- The binary search tree for which the average number of comparisons in a search is the smallest as possible is called as optimal binary search tree.
- If probabilities of searching for elements of a set are known an optimal binary search tree can be constructed.

Optimal Binary Search Tree

Example

- Consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively.
- 14 different binary search trees are possible
- Two are shown here



- The average number of comparisons in a successful search in the first of these trees is

  0.1 *1 + 0.2 * 2 + 0.4 *3+ 0.3 *4 = 2.9,

  0.1 * 2 + 0.2 * 1+ 0.4 * 2 + 0.3 *3= 2.1.

  Neither of these two trees is, in fact, optimal.

Optimal BST

- For our tiny example, we could find the optimal tree by generating all 14 binary search trees with these keys.
- As a general algorithm, this exhaustive-search approach is unrealistic:
  - the total number of binary search trees with n keys is equal to the nth ***Catalan*** number,

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$

which grows to infinity as fast as $4^n / n^{1.5}$

Optimal BST

Problem definition:

- Given a sorted array $a_1, \ldots, a_n$ of search keys and an array $p_1, \ldots, p_n$ of probabilities of searching, where p_i is the probability of searches to a_i.
- Construct a binary search tree of all keys such that, smallest average number of comparisons made in a successful search.

Optimal BST – Solution using DP

- So let $a_1, \ldots, a_n$ be distinct keys ordered from the smallest to the largest and let $p_1, \ldots, p_n$ be the probabilities of searching for them.
- Let **C(i, j)** be the smallest average number of comparisons made in a successful search in a binary search tree $\mathbf{T}_i^j$ made up of keys $a_i, \ldots, a_j$, where i, j are some integer indices, $1 \leq i \leq j \leq n$.
- We are interested just in **C(1, n).**

Optimal BST – Solution using DP

- To derive a recurrence, we will consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$.

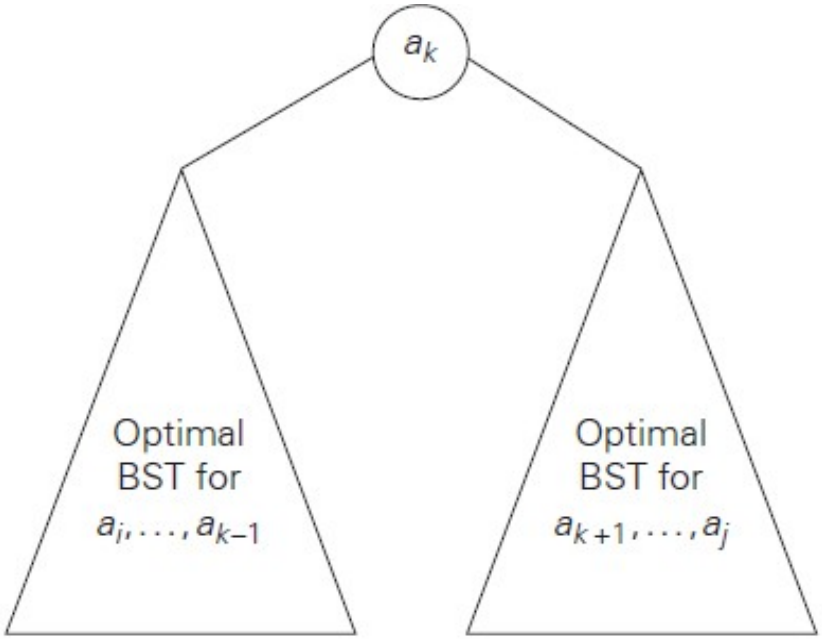
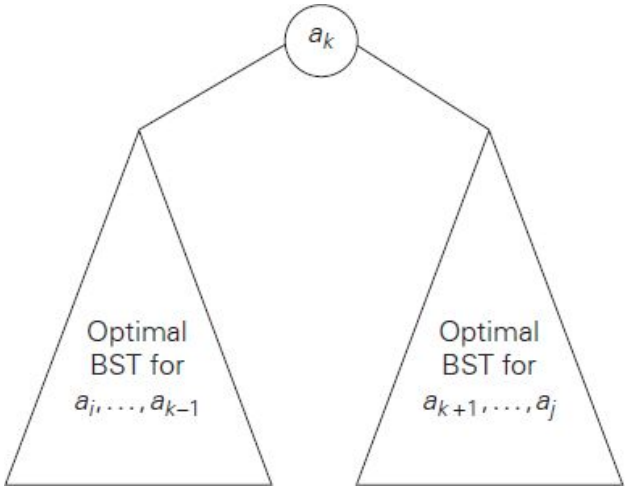


FIGURE 8.8 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^{j}.

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n.$$



We assume in formula (8.8) that $C(i, i-1) = 0$ for $1 \le i \le n+1$, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that

$$C(i, i) = p_i \quad \text{for } 1 \le i \le n,$$

as it should be for a one-node binary search tree containing a_i.

Example

index	1	2	3	4
key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n.$$

C(i,j)	0	1	2	3	4
1	0				
2		0			
3			0		
4				0	
5					0

C(i,j)	0	1	2	3	4
1	0	0.1 1			
2		0	0.2 2		
3			0	0.4 3	
4				0	0.3 4
5					0

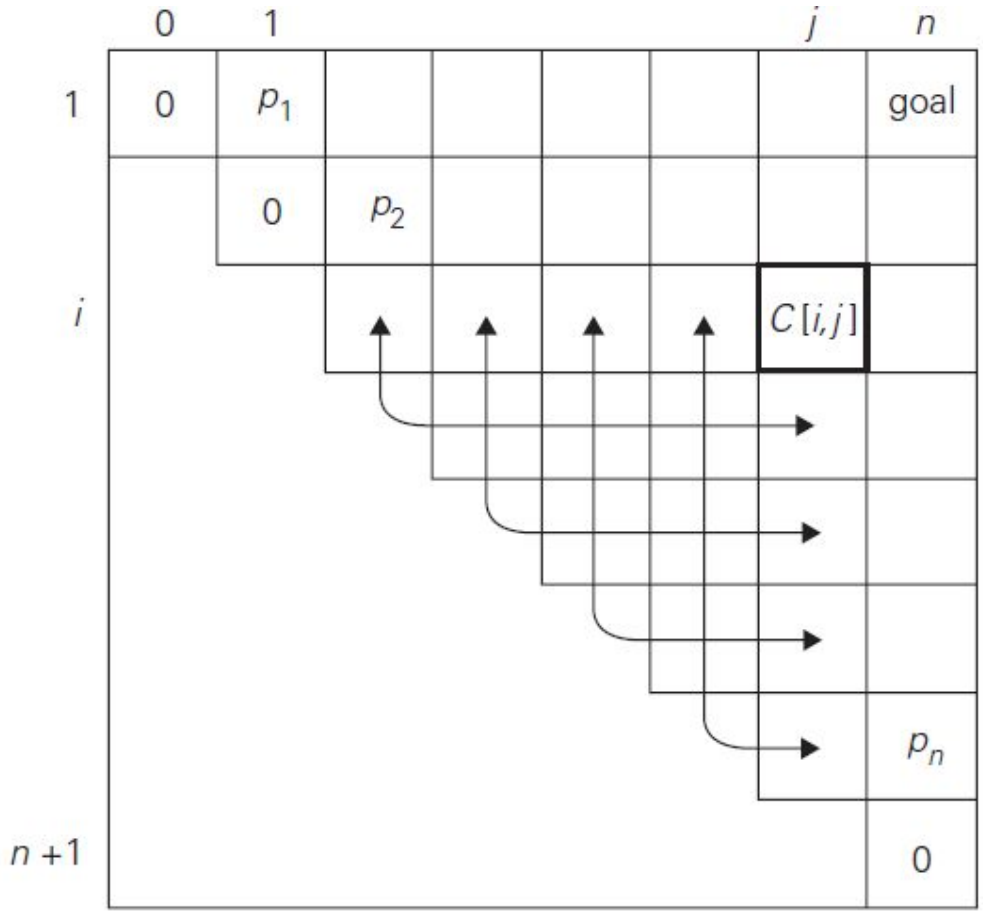
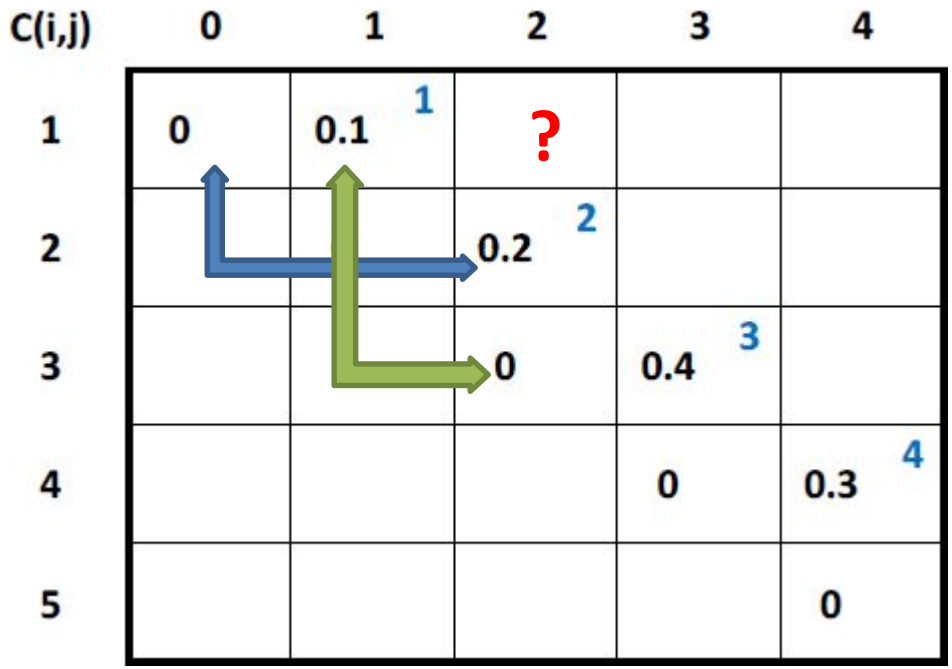


FIGURE 8.9 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

Example

index	1	2	3	4
key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3



$$C(1,2) = \text{Min} \begin{Bmatrix} k = 1,\ C(1,0) + C(2,2), \\ k = 2,\ C(1,1) + C(3,2) \end{Bmatrix} + 0.3 = \text{Min} \begin{Bmatrix} 0.2 \\ 0.1 \end{Bmatrix} + 0.3 = 0.4$$

$$C(i,j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n.$$

Example

index	1	2	3	4
key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

C(i,j)	0	1	2	3	4
1	0	0.1 (1)			
2		0	0.2 (2)		
3			0	0.4 (3)	
4				0	0.3 (4)
5					0

Example

index	1	2	3	4
key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

C(i,j)	0	1	2	3	4
1	0	0.1 1	0.4 2		
2		0	0.2 2	0.8 3	
3			0	0.4 3	1.0 3
4				0	0.3 4

Example

index	1	2	3	4
key	*A*	*B*	*C*	*D*
probability	0.1	0.2	0.4	0.3

C(i,j)	0	1	2	3	4
1	0	0.1 1	0.4 2	1.1 3	1.7 3
2		0	0.2 2	0.8 3	1.4 3
3			0	0.4 3	1.0 3
4				0	0.3 4
5					0

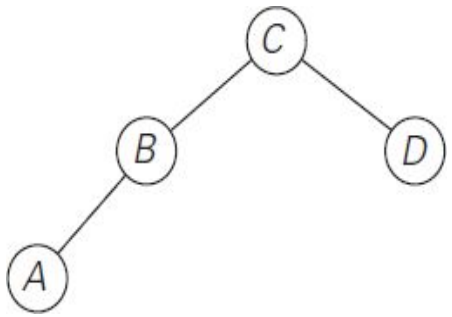
How to construct the optimal binary search tree?

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

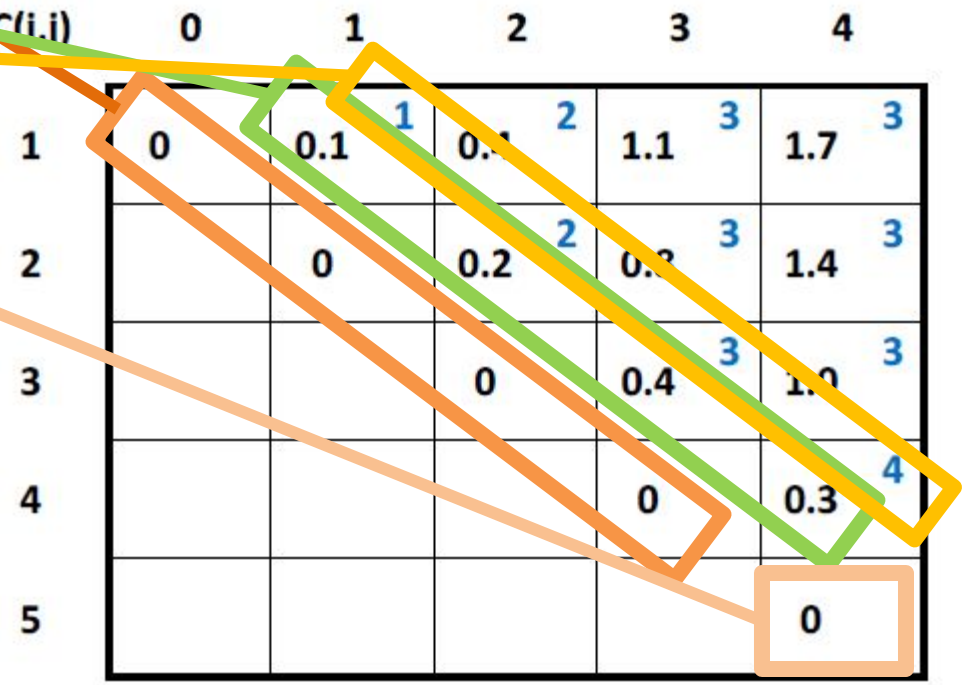
	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					



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 $i \leftarrow 1$ **to** n **do**
  $C[i, i-1] \leftarrow 0$
  $C[i, i] \leftarrow P[i]$
  $R[i, i] \leftarrow i$
$C[n+1, n] \leftarrow 0$



$$C(i, j) = \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n.$$

C(i,j)	0	1	2	3	4
1	0	0.1 (1)	0.4 (2)	1.1 (3)	1.7 (3)
2		0	0.2 (2)	0.8 (3)	1.4 (3)
3			0	0.4 (3)	1.0 (3)
4				0	0.3 (4)
5					0

for $d \leftarrow 1$ **to** $n-1$ **do** //diagonal

  for $i \leftarrow 1$ **to** $n-d$ **do**

    $j \leftarrow i+d$

$minval \leftarrow \infty$

for $k \leftarrow i$ **to** j **do**

  if $C[i, k-1] + C[k+1, j] < minval$

    $minval \leftarrow C[i, k-1] + C[k+1, j]; \quad kmin \leftarrow k$

$R[i, j] \leftarrow kmin$

$sum \leftarrow P[i]; \quad$ **for** $s \leftarrow i+1$ **to** j **do** $sum \leftarrow sum + P[s]$

$C[i, j] \leftarrow minval + sum$

return $C[1, n]$, R

Analysis

- The algorithm requires **$O(n^2)$** time and **$O(n^2)$** storage.
- Therefore, as ‘n’ increases it will run out of storage even before it runs out of time.
- The storage needed can be reduced by almost half by implementing the two-dimensional arrays as one- dimensional arrays.

Home Work

- Find the optimal binary search tree for the keys **A, B, C, D, E** with search probabilities **0.1, 0.1, 0.2, 0.2, 0.4** respectively.

Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- Transitive Closure
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- Reliability design

Knapsack problem using DP

- Given **n** items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.
- To design a DP algorithm, we need to derive a recurrence relation that expresses a solution in terms of its smaller sub instances.
- Let us consider an instance defined by the first i items, $1 \leq i \leq n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \leq j \leq W$.

Knapsack problem using DP

- Let **F(i, j)** be the value of an **optimal solution** to this instance.
- We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories:
  - those that do not include the i^{th} item
  - and those that do.
- Among the subsets that **do not include the i^{th} item**,
  - the value of an optimal subset is, by definition,
  - i.e F(i , j) = F(i − 1, j).

Knapsack problem using DP

- Among the subsets that **do include the i^{th} item** (hence, $j - w_i \geq 0$),
  - an optimal subset is made up of this item and
  - an optimal subset of the first i−1 items that fits into the knapsack of capacity $j - w_i$.
  - The value of such an optimal subset is $F(i, j) = v_i + F(i - 1, j - w_i)$

- Thus, optimal solution among all feasible subsets of the first I items is the maximum of these two values

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

Knapsack problem using DP

- It is convenient to define the initial conditions as follows: F(0, j) = 0 for j ≥ 0 and F(i, 0) = 0 for i ≥ 0.

- Our goal is to find **F(n, W),** the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

		0	$j-w_j$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_j)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Table for solving the knapsack problem by dynamic programming.

Algorithm

```
for w = 0 to W
    F[0,w] = 0
for i = 1 to n
    F[i,0] = 0
for i = 1 to n
    for w = 0 to W
        if w_i <= w // item i can be part of the solution
            if v_i + F[i-1,w-w_i] > F[i-1,w]
                F[i,w] = v_i + F[i-1,w- w_i]
            else
                F [i,w] = F [i-1,w]
        else F [i,w] = F [i-1,w]  // w_i > w
```

Example-1

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

capacity $W = 5$.

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

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

Find the composition of an optimal subset by **backtracing** the computations

Example-1

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

capacity $W = 5$.

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

		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**

Find the composition of an optimal subset by **backtracing** the computations

Analysis

- The classic dynamic programming approach, works bottom up: it fills a table with solutions to all smaller subproblems, each of them is solved only once.
- Drawback: Some unnecessary subproblems are also solved

- The time efficiency and space efficiency of this algorithm are both in **Θ(nW).**
- The time needed to find the composition of an optimal solution is in **O(n).**

Discussion

- The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient.
- Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches.
- The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using **memory functions**.

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 capacity
- **Output:** The value of an optimal feasible subset of the first i items
- **Note:** Uses as global variables input arrays **Weights[1..n], V alues[1..n],** and **table F[0..n, 0..W]** whose entries are initialized with −1's except for row 0 and column 0 initialized
  with 0's

Algorithm MFKnapsack(i, j)

```
if F[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]))
    F[i, j] ← value
return F[i, j]
```

Example

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

		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**

Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- Transitive Closure:
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- **Bellman-For d Algorithm**
- Travelling Sales Person problem
- Reliability design

Single Source shortest path with –ve wts

Problem definition

- Single source shortest path - Given a graph and a source vertex ***s*** in graph, find shortest paths from ***s*** to all vertices in the given graph.
- The graph may contain negative weight edges.

- Dijkstra's algorithm [O(VlogV)], doesn't work for graphs with negative weight edges.

- Bellman-Ford works in O(VE)

Bellman Ford Algorithm

- Like other Dynamic Programming Problems, the algorithm calculates shortest paths in bottom-up manner.
- It first calculates the shortest distances
  - for the shortest paths which have at-most one edge in the path.
  - Then, it calculates shortest paths with at-most 2 edges, and so on.

- Iteration i finds all shortest paths that use i edges.

Let $dist^{\ell}[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most ℓ edges. Then, $dist^{1}[u] = cost[v, u]$, $1 \leq u \leq n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n - 1$ edges. Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u.

1. If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k - 1$ edges, then $dist^{k}[u] = dist^{k-1}[u]$.

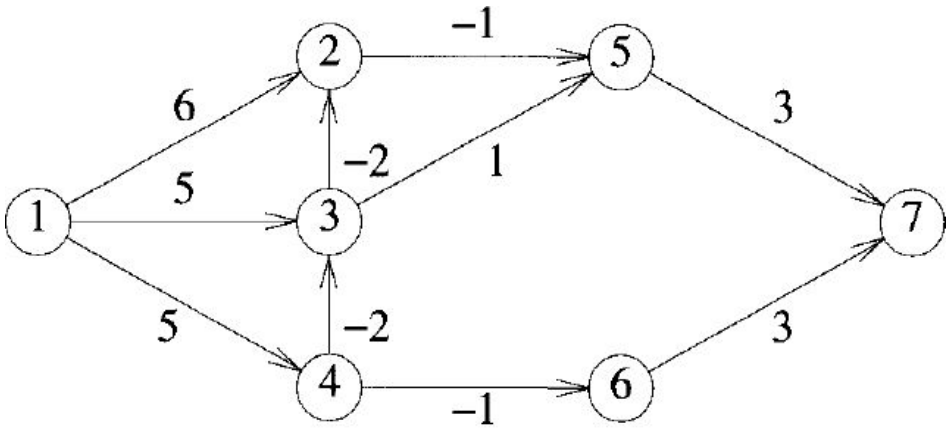
2. If the shortest path from v to u with at most k, $k > 1$, edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the edge $\langle j, u \rangle$. The path from v to j has $k - 1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $\langle i, u \rangle$ is in the graph are candidates for i. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i, u]$ is the correct value for j.

These observations result in the following recurrence for *dist*:

$$dist^k[u] = \min\{dist^{k-1}[u], \min_i\{dist^{k-1}[i] + cost[i,u]\}\}$$

This recurrence can be used to compute $dist^k$ from $dist^{k-1}$, for $k = 2, 3, \ldots, n-1$.

```
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];
}
```



(a) A directed graph

$dist^1(1) = 0$
$dist^1(2) = 6$
$dist^1(3) = 5$
$dist^1(4) = 5$
$dist^1(5) = \infty$
$dist^1(6) = \infty$
$dist^1(7) = \infty$

k=1

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

(b) $dist^k$

For every vertex (except source) look for the incoming edge (except from source)

$dist^2(1) = 0$

$dist^2(2) = \min \begin{Bmatrix} dist^1(2) \\ dist^1(3) + cost(3,2) \end{Bmatrix} = \min \begin{Bmatrix} 6 \\ 5 - 2 \end{Bmatrix} = 3$

$dist^2(3) = \min \begin{Bmatrix} dist^1(3) \\ dist^1(4) + cost(4,3) \end{Bmatrix} = \min \begin{Bmatrix} 5 \\ 5 - 2 \end{Bmatrix} = 3$

$dist^2(4) = 5$

$dist^2(5) = \min \begin{Bmatrix} dist^1(5) & \infty \\ dist^1(2) + cost(2,5) & \end{Bmatrix} = \min 6 \begin{Bmatrix} 1 \\ 5 + 1 \end{Bmatrix} = 5$

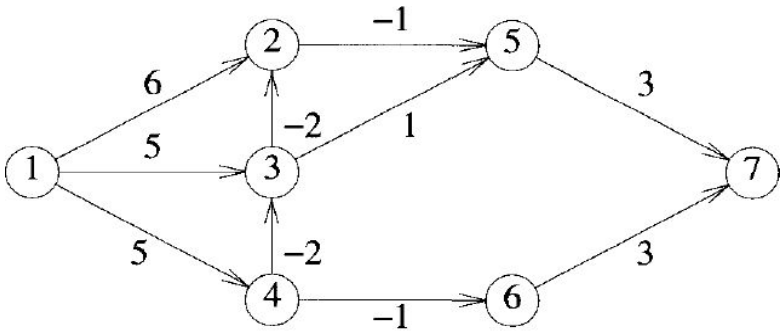
$dist^2(6) = \min \begin{Bmatrix} dist^1(3) + cost(3,5) \\ dist_1(6) \end{Bmatrix} = \min \begin{Bmatrix} & \infty \end{Bmatrix} = 4$

$dist^2(7) = \min \begin{Bmatrix} dist^1(4) + cost(4,6) \\ dist^1(7) \end{Bmatrix} \begin{matrix} 5 - 1 \\ \infty \end{matrix} \begin{Bmatrix} \\ \end{Bmatrix}$

$dist^1(5) + cost(5,7) = \min \begin{matrix} \infty + 3 \\ \infty + 3 \end{matrix} = \infty$

$dist^1(6) + cost(6,7)$

k=2



(a) A directed graph

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

(b) $dist^k$

For every vertex (except source) look for the incoming edge (except from source)

$dist^3(1) = 0$

$$dist^3(2) = \min \begin{Bmatrix} dist^2(2) \\ dist^2(3) + cost(3,2) \end{Bmatrix} = \min \begin{Bmatrix} 3 \\ 3 - 2 \end{Bmatrix} = 1$$

$$dist^3(3) = \min \begin{Bmatrix} dist^2(3) \\ dist^2(4) + cost(4,3) \end{Bmatrix} = \min \begin{Bmatrix} 3 \\ 5 - 2 \end{Bmatrix} = 3$$

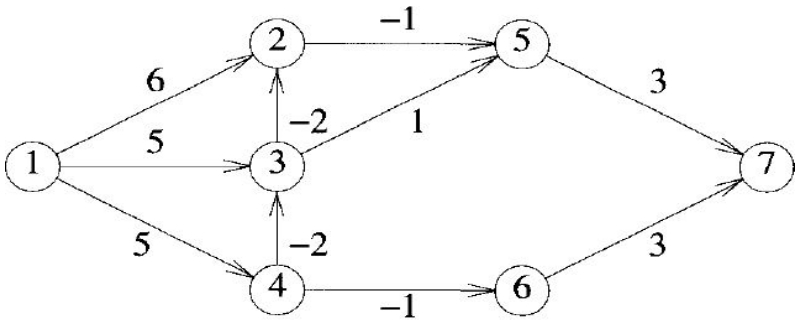
$dist^3(4) = 5$

$$dist^3(5) = \min \begin{Bmatrix} dist^2(5) \\ dist^2(2) + cost(2,5) \\ dist^2(3) + cost(3,5) \end{Bmatrix} = \min \begin{Bmatrix} 5 \\ 3-1 \\ 3 + 1 \end{Bmatrix} = 2$$

$$dist^3(6) = \min \begin{Bmatrix} dist^2(6) \\ dist^2(4) + cost(4,6) \end{Bmatrix} = \min \begin{Bmatrix} 4 \\ 5 - 1 \end{Bmatrix} = 4$$

$$dist^3(7) = \min \begin{Bmatrix} dist^2(7) \\ dist^2(5) + cost(5,7) \\ dist^2(6) + cost(6,7) \end{Bmatrix} = \min \begin{Bmatrix} \infty \\ 7 \\ 4 + 3 \end{Bmatrix} = 5 + 3 =$$

k=3



(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							
5							
6							

(b) $dist^k$

For every vertex (except source) look for the incoming edge (except from source)

$$\text{dist}^4(1) = 0$$

$$\text{dist}^4(2) = \min \left\{ \begin{array}{c} \text{dist}^3(2) \\ \text{dist}^3(3) + \text{cost}(3,2) \end{array} \right\} = \min \left\{ \begin{array}{c} 1 \\ 3 - 2 \end{array} \right\} = 1$$

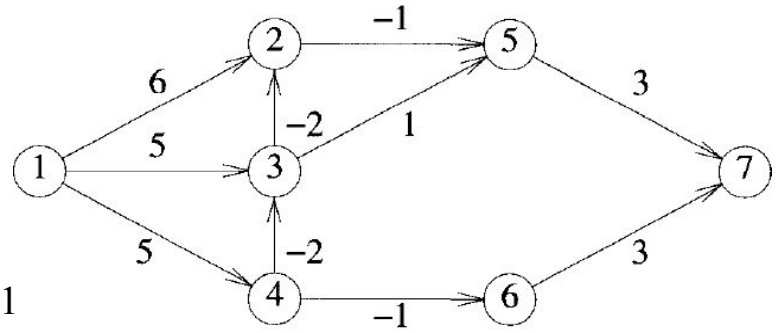
$$\text{dist}^4(3) = \min \left\{ \begin{array}{c} \text{dist}^3(3) \\ \text{dist}^3(4) + \text{cost}(4,3) \end{array} \right\} = \min \left\{ \begin{array}{c} 3 \\ 5 - 2 \end{array} \right\} = 3$$

$$\text{dist}^4(4) = 5$$

$$\text{dist}^4(5) = \min \left\{ \begin{array}{l} \text{dist}^3(5) \\ \text{dist}^3(2) + \text{cost}(2,5) \\ \text{dist}^3(3) + \text{cost}(3,5) \end{array} \right\} = \min \left\{ \begin{array}{l} 2 \\ 1 - 1 \\ 3 + 1 \end{array} \right\} = 0$$

$$\text{dist}^4(6) = \min \left\{ \begin{array}{c} \text{dist}^3(6) \\ \text{dist}^3(4) + \text{cost}(4,6) \end{array} \right\} = \min \left\{ \begin{array}{c} 4 \\ 5 - 1 \end{array} \right\} = 4$$

$$\text{dist}^4(7) = \min \left\{ \begin{array}{l} \text{dist}^3(7) \\ \text{dist}^3(5) + \text{cost}(5,7) \\ \text{dist}^3(6) + \text{cost}(6,7) \end{array} \right\} = \min \left\{ \begin{array}{l} 7 \\ 2 + 3 \\ 4 + 3 \end{array} \right\} = 5$$



(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							
6							

(b) $dist^k$

k=4

For every vertex (except source) look for the incoming edge (except from source)

$$\text{dist}^5(1) = 0$$

$$\text{dist}^5(2) = \min \left\{ \begin{array}{c} \text{dist}^4(2) \\ \text{dist}^4(3) + \text{cost}(3,2) \end{array} \right\} = \min \left\{ \begin{array}{c} 1 \\ 3 - 2 \end{array} \right\} = 1$$

$$\text{dist}^5(3) = \min \left\{ \begin{array}{c} \text{dist}^4(3) \\ \text{dist}^4(4) + \text{cost}(4,3) \end{array} \right\} = \min \left\{ \begin{array}{c} 3 \\ 5 - 2 \end{array} \right\} = 3$$

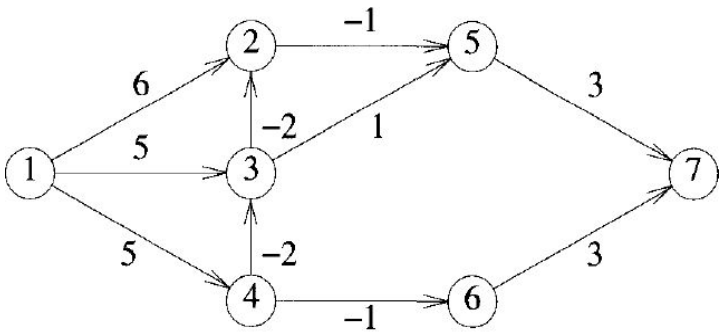
$$\text{dist}^5(4) = 5$$

$$\text{dist}^5(5) = \min \left\{ \begin{array}{l} \text{dist}^4(5) \\ \text{dist}^4(2) + \text{cost}(2,5) \\ \text{dist}^4(3) + \text{cost}(3,5) \end{array} \right\} = \min \left\{ \begin{array}{l} 0 \\ 1 - 1 \\ 3 + 1 \end{array} \right\} = 0$$

$$\text{dist}^5(6) = \min \left\{ \begin{array}{c} \text{dist}^4(6) \\ \text{dist}^4(4) + \text{cost}(4,6) \end{array} \right\} = \min \left\{ \begin{array}{c} 4 \\ 5 - 1 \end{array} \right\} = 4$$

$$\text{dist}^5(7) = \min \left\{ \begin{array}{l} \text{dist}^4(7) \\ \text{dist}^4(5) + \text{cost}(5,7) \\ \text{dist}^4(6) + \text{cost}(6,7) \end{array} \right\} = \min \left\{ \begin{array}{l} 5 \\ \\ 0 + 3 \\ 4 + 3 \end{array} \right\} = 3$$

k=5

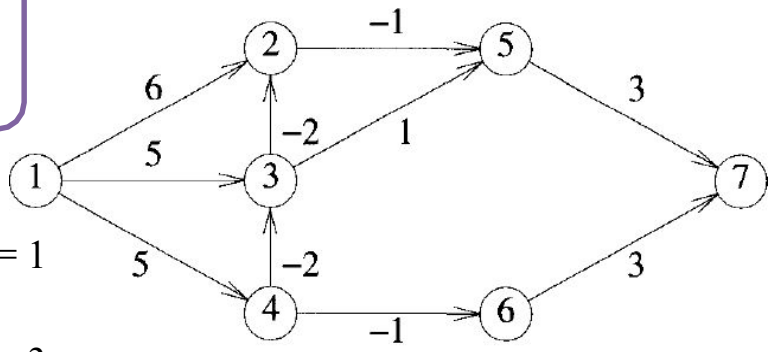


(a) A directed graph

	$dist^k[1..7]$						
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							
6							

(b) $dist^k$

For every vertex (except source) look for the incoming edge (except from source)



(a) A directed graph

$$dist^6(1) = 0$$

$$dist^6(2) = \min \left\{ \begin{array}{c} dist^5(2) \\ dist^5(3) + cost(3,2) \end{array} \right\} = \min \left\{ \begin{array}{c} 1 \\ 3 - 2 \end{array} \right\} = 1$$

$$dist^6(3) = \min \left\{ \begin{array}{c} dist^5(3) \\ dist^5(4) + cost(4,3) \end{array} \right\} = \min \left\{ \begin{array}{c} 3 \\ 5 - 2 \end{array} \right\} = 3$$

$$dist^6(4) = 5$$

$$dist^6(5) = \min \left\{ \begin{array}{c} dist^5(5) \\ dist^5(2) + cost(2,5) \\ dist^5(3) + cost(3,5) \end{array} \right\} = \min \left\{ \begin{array}{c} 0 \\ 1 - 1 \\ 3 + 1 \end{array} \right\} = 0$$

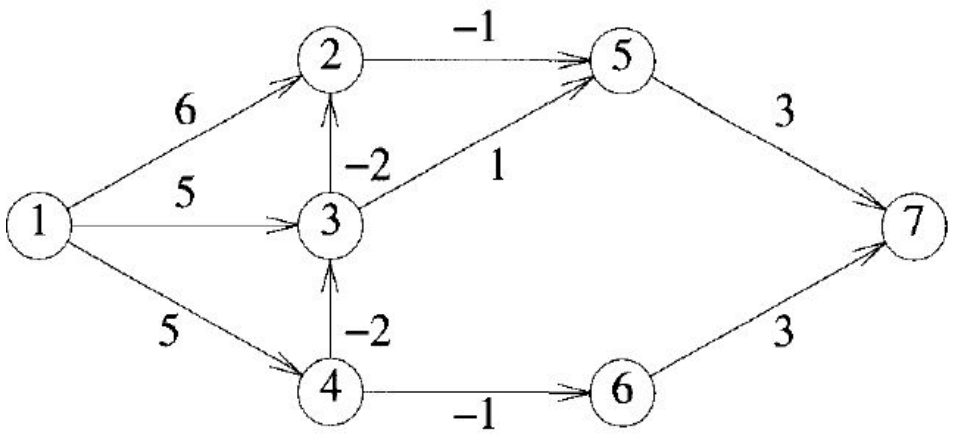
$$dist^6(6) = \min \left\{ \begin{array}{c} dist^5(6) \\ dist^5(4) + cost(4,6) \end{array} \right\} = \min \left\{ \begin{array}{c} 4 \\ 5 - 1 \end{array} \right\} = 4$$

$$dist^6(7) = \min \left\{ \begin{array}{c} dist^5(7) \\ dist^5(5) + cost(5,7) \\ dist^5(6) + cost(6,7) \end{array} \right\} = \min \left\{ \begin{array}{c} 5 \\ 0 + 3 \\ 4 + 3 \end{array} \right\} = 3$$

	$dist^k[1..7]$						
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							

(b) $dist^k$

k=6



(a) A directed graph

k	$dist^k[1..7]$						
	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) $dist^k$

Algorithm

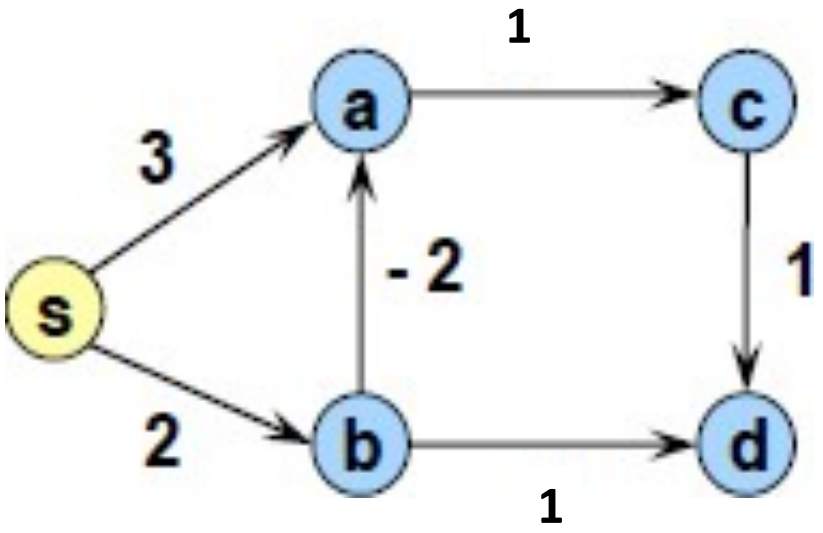
```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
     for i := 1 to n do // Initialize dist.
          dist[i] := cost[v, i];
     for k := 2 to n − 1 do
          for each u such that u ≠ v and u has
                    at least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

Analysis

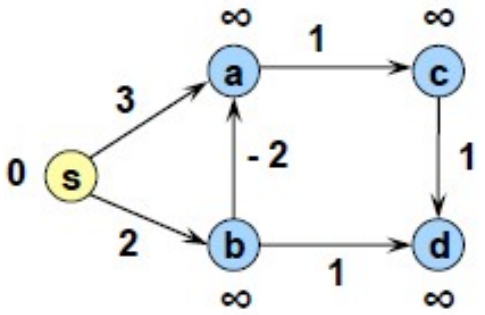
- Bellman-Ford works in **O(VE)**

Example-2 Home work

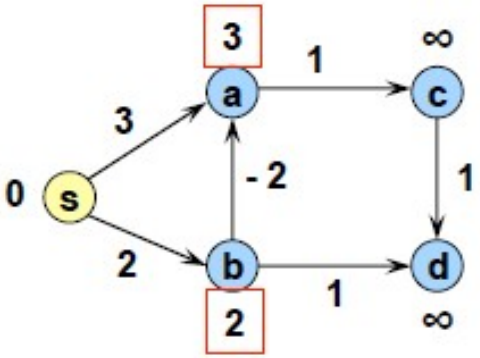
Find shortest path from S to all other vertices using Bellman Ford algorithm



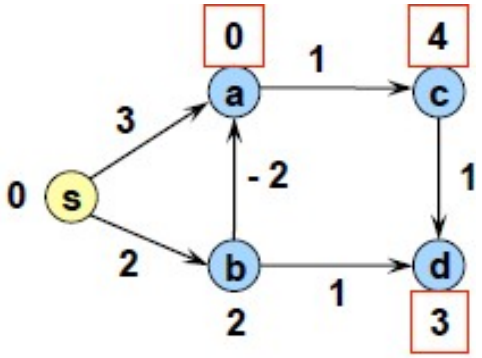
Example-2



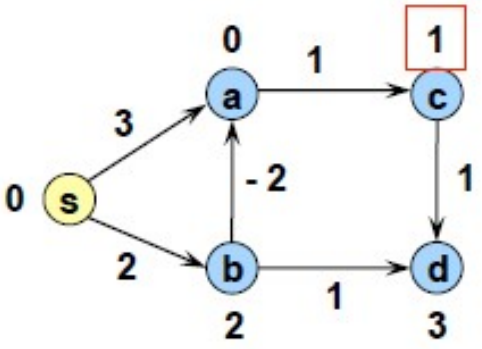
path lengths = 0



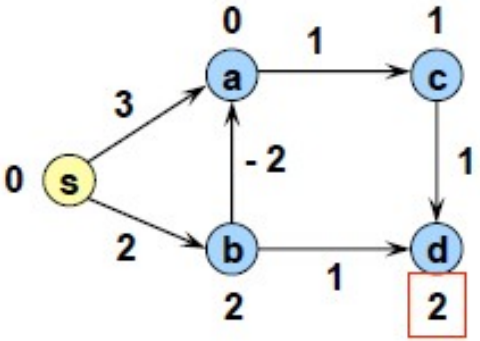
path lengths ≤ 1



path lengths ≤ 2



path lengths ≤ 3



path lengths ≤ 4

Outline

- Introduction to Dynamic Programming
  - General method with Examples
  - Multistage Graphs
- Transitive Closure:
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- **Travelling Sales Person problem**
- Reliability design

Travelling Salesman Problem

Problem Statement

- The travelling salesman problem consists of a salesman and a set of cities.
- The salesman has to visit each one of the cities exactly once starting from a certain one (e.g. the hometown) and returning to the same city.
- The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip
- It is assumed that all cities are connected to all other cities

Travelling Salesman Problem

Naive Solution:

1. Consider city 1 as the starting and ending point.
2. Generate all (n-1)! Permutations of cities.
3. Calculate cost of every permutation and keep track of minimum cost permutation.
4. Return the permutation with minimum cost.

Time Complexity: ?

$n!$

TSP using DP

- Assume tour to be a simple path that starts & ends at 1
- Every tour consists of
  - an edge <1, k> for some k $\in$ V – {1} and
  - a path from vertex k to vertex 1
- The path from k to 1 goes through each vertex in V – {1, k} exactly once
- If the tour is optimal, then the path from k to 1 must be a shortest path going through all vertices in V – {1, 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
- g(1, V – {1}) is the length of an optimal salesperson tour

TSP using DP

- From the principal of optimality

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

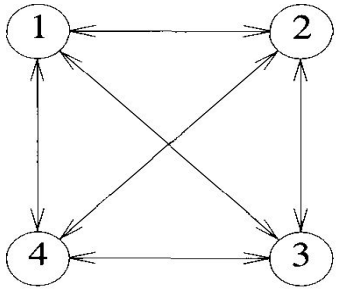
- Generalizing

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

- Example

$$g(1,\{2,3,4,5\}) = \min \begin{cases} k = 2, c_{12} + g(2,\{3,4,5\}) \\ k = 3, c_{13} + g(3,\{2,4,5\}) \\ k = 4, c_{14} + g(4,\{2,3,5\}) \\ k = 5, c_{15} + g(5,\{2,3,4\}) \end{cases}$$

Example-1



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

(b)

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

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

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

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

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

Successor of node 1: p(1,{2,3,4}) = 2
Successor of node 3: p(2, {3,4}) = 4
Successor of node 4: p(4, {3}) = 3

Optimal tour

Example-2 (Home Work)

$$C = \begin{pmatrix} 0 & 2 & 9 & 10 \\ 1 & 0 & 6 & 4 \\ 15 & 7 & 0 & 8 \\ 6 & 3 & 12 & 0 \end{pmatrix}$$

$g(2, \emptyset) = c_{21} = 1$
$g(3, \emptyset) = c_{31} = 15$
$g(4, \emptyset) = c_{41} = 6$

k = 1, consider sets of 1 element:

Set {2}:
$g(3,\{2\}) = c_{32} + g(2,\emptyset) = c_{32} + c_{21} = 7 + 1 = 8$ $\quad p(3,\{2\}) = 2$
$g(4,\{2\}) = c_{42} + g(2,\emptyset) = c_{42} + c_{21} = 3 + 1 = 4$ $\quad p(4,\{2\}) = 2$

Set {3}:
$g(2,\{3\}) = c_{23} + g(3,\emptyset) = c_{23} + c_{31} = 6 + 15 = 21$ $\quad p(2,\{3\}) = 3$
$g(4,\{3\}) = c_{43} + g(3,\emptyset) = c_{43} + c_{31} = 12 + 15 = 27$ $\quad p(4,\{3\}) = 3$

Set {4}:
$g(2,\{4\}) = c_{24} + g(4,\emptyset) = c_{24} + c_{41} = 4 + 6 = 10$ $\quad p(2,\{4\}) = 4$
$g(3,\{4\}) = c_{34} + g(4,\emptyset) = c_{34} + c_{41} = 8 + 6 = 14$ $\quad p(3,\{4\}) = 4$

Complete the exercise….

Analysis

An algorithm that proceeds to find an optimal tour by using (5.20) and (5.21) will require $\Theta(n^2 2^n)$ time as the computation of $g(i, S)$ with $|S| = k$ requires $k - 1$ comparisons when solving (5.21). This is better than enumerating all $n!$ different tours to find the best one. The most serious drawback of this dynamic programming solution is the space needed, $O(n2^n)$. This is too large even for modest values of n.

Outline

- Introduction to Dynamic P
  - General method with Exam
  - Multistage Graphs
- Transitive Closure:
  - Warshall's Algorithm,
- All Pairs Shortest Paths:
  - Floyd's Algorithm,
- Optimal Binary Search Trees
- Knapsack problem
- Bellman-Ford Algorithm
- Travelling Sales Person problem
- **Reliability design**

Reliability Design

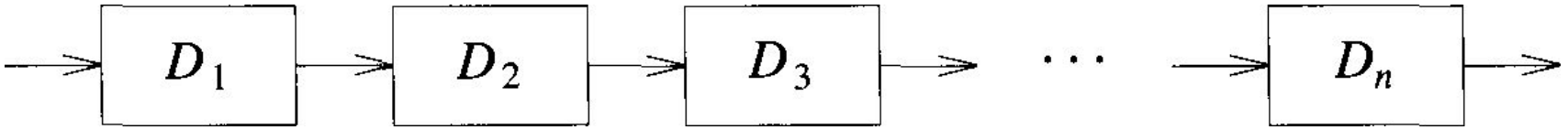


Figure 5.19 n devices D_i, $1 \leq i \leq n$, connected in series

- If n devices D_i, $1 \leq i \leq n$, connected in series the reliability of whole system is

$$\prod_{i=1}^{n} r_i$$

Reliability Design

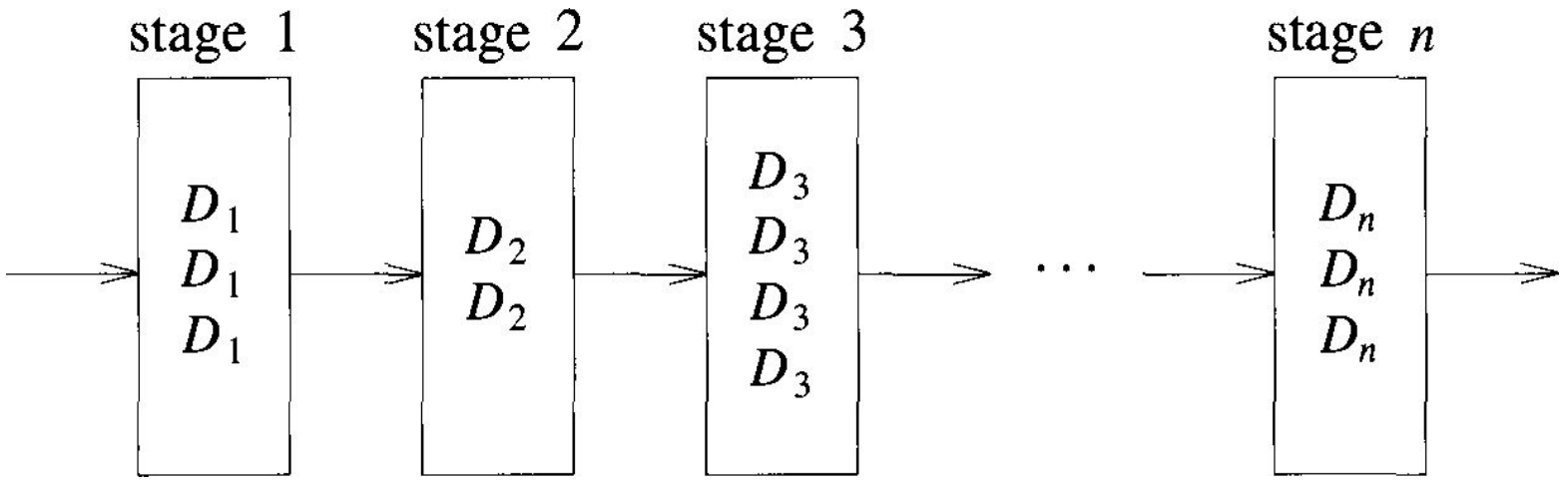


Figure 5.20 Multiple devices connected in parallel in each stage

- Let r_i be the reliability of the device at stage i.
- If stage i has m_i devices of type i in parallel then reliability of stage i is

$$\phi(i, m_i) = 1 - (1 - r_i)^{m_i}$$

Example

- Design a 3-stage system with device types A,B,C. There costs are 30, 15, 20 and reliability are 0.9, 0.8,
  0.5 respectively. Budget available is 105. Design a system with highest reliability.

Devices	A	B	C
cost	30	15	20
Reliability	0.9	0.8	0.5
Maximum number of devices can be purchased (by guaranteed purchase of one unit of other devices)			

Budget =105

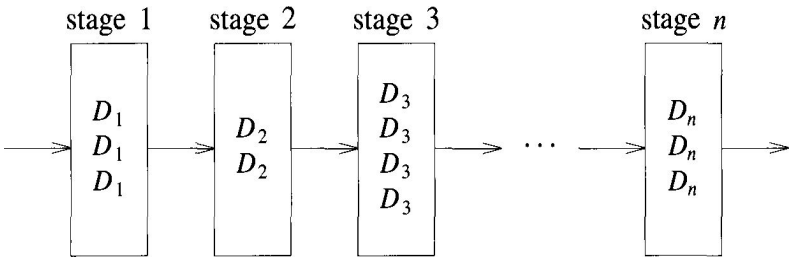
Example

Budget =105

$$\varphi_i(m_i) = 1 - (1 - r_i)^{m_i}$$

No. of devices in llel	A		B		C	
	Reliability	Cost	Reliability	Cost	Reliability	Cost
1	0.9 m1=1	30	0.8 m2=1	15	0.5 m3=1	20
2	0.99 m1=2	60	0.96 m2=2	30	0.75 m3=2	40
3	-	-	0.992 m2=3	45	0.875 m3=3	60

- S^i - Set of all tuples of the from *(f,x)* generated from various sequences m1, m2.. mi considering i stages where f is reliability and x is cost
- S^i_j - i is the stage no., j is the no. of devices used at stage i
- Example
  - S^1_1 means 1 device of type A at stage 1
  - S^1_2 means 2 device of type A at stage 1



# of devices in llel	A		B		C	
	Reliability	Cost	Reliability	Cost	Reliability	Cost
1	0.9 m1=1	30	0.8 m2=1	15	0.5 m3=1	20
2	0.99 m1=2	60	0.96 m2=2	30	0.75 m3=2	40
3	-	-	0.992 m2=3	45	0.875 m3=3	60

$S_1^1 = \{(0.9, 30)\}$

$S_2^1 = \{(0.99, 60)\}$

Combining: $S^1 = \{(0.9, 30), (0.99, 60)\}$ (m1=1, m1=2)

.9*.8 .99*.8

$S_1^2 = \{(0.72, 45), (0.792, 75)\}$

.9*.96 .99*.96

$S_2^2 = \{(0.864, 60)\}$

.9*.992 .99*.992

$S_3^2 = \{(0.8928, 75)\}$

For (0.9504,90) refer Note #1

For (0.982,105) refer Note #1

Note #2

Combining: $S^2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75)\}$

$S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

m1=1 m2=1 m1=1 m2=2 m1=1 m2=3

Note #1: This configuration will not leave adequate funds to complete system

Note #2: For the same cost, retain the one w

# of devices in llel	A		B		C	
	Reliability	Cost	Reliability	Cost	Reliability	Cost
1	0.9 m1=1	30	0.8 m2=1	15	0.5 m3=1	20
2	0.99 m1=2	60	0.96 m2=2	30	0.75 m3=2	40
3	-	-	0.992 m2=3	45	0.875 m3=3	60

$S^2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

m1=1 m2=1 | m1=1 m2=2 | m1=1 m2=3

$S_1^3 = \{(.36, 65), (.432, 80), (.4464, 95)\}$

$S_2^3 = \{(.54, 85), (.648, 100)\}$

$S_3^3 = \{(.63, 105)\}$

m1=1 m2=1 M3 =3

Combining: $S^3 = \{(.36, 65), (.432, 80), (.54, 85), (.648, 100), (.4464, 95), (.63, 105)\}$

Best Reliability 0.648 with cost=100 where A -1 unit, B -1 unit, C- 2 units in ||el.

Assignment-4

1. Write multistage graph algorithm using backward approach.
2. Define transitive clossure of a directed graph. Find the transitive clossure matrix for the graph whose adjacency matrix is given.

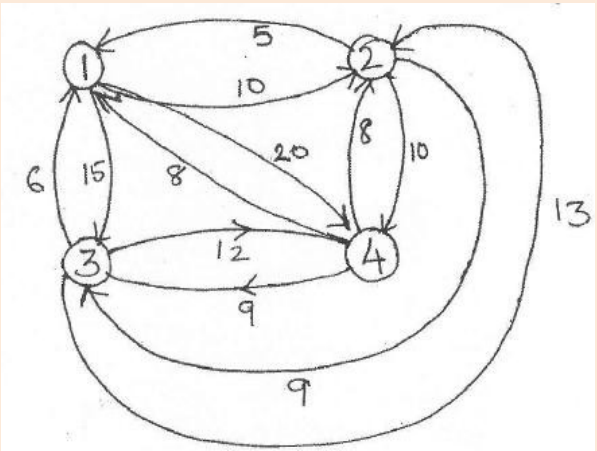
$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

3. Apply bottom up dynamic programming algorithm for the following instance of the knapsack problem. Knapsack capacity = 10.

Item	Weight	Value
1	7	42
2	3	12
3	4	40
4	5	25

Assignment-4

4. For the given graph obtain optimal cost tour using dynamic programming.



5. Apply Floyds algorithm to find the all pair shortest path for the graph given below.