# UNIT - 4DYNAMIC PROGRAMMING

## UNIT STRUCTURE



## 4.1 LEARNINGOBJECTIVES

After goingthrough this unit, you will be able to:

- understand the concept of Dynamic Programming
- solve problems using dynamic programming approach
- get familiarize with optimality conditions

## 4.2 INTRODUCTION

Intheprecedingunits,wehaveseensomeelegantdesignprinciplessuch asdivide-and-conquer, greedy algorithm-thatyielddefinitivealgorithmsfor avariety ofimportant computationaltasks.Thedrawbackofthesetechniquesisthattheycanonlybeusedonveryspecific typesofproblems.In this unit, we will introduce you the dynamic programming technique. We will concentrate on elaborating 0/1 Knapsack problem and travelling salesman problem in this unit.

## 4.3GENERAL STRATEGY

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. A problem issaid to possess an optimal substructure if an optimal solution to the problem contains within the optimal solutions of its sub-problems. Whena problem exhibits optimal substructure, is a good clue that dynamic programming might apply. In dynamic programming, we build an optimal solution to the problem from optimal solutions of its sub-problems. Consequently, we must take care that, the range of sub-problems we consider includes those sub-problemswhich are used in the optimal solution.Some important concept of dynamic programming are :

**Stage of a Problem**

The dynamic programming problem can be divided in to a sequence of smaller sub-problems called stages of the original problem.

**State of a Problem**

The condition of decision process at a stage is called its state. The decision variable which specify the condition of decision process at a particular stage is called state variable.

**Principle of Optimality**

A problem is said to satisfy the Principle of Optimality if the sub-solutions of an optimal solution of the problem are themselves optimal solutions for their sub-problems. For examples: The shortest path problem satisfies the Principle of Optimality. This is because if a,x1,x2,...,xn,b is a shortest path from node a to node b in a graph, then the portion of xi to xj on that path is a shortest path from xi to xj.

**Characteristics of Dynamic Programming**

i) The Problem can be divided into stages, with a policy decision at each stage
ii) Each stage consist of a number of states associated with it
iii) Decision at each stage convert the current stage in to a state associated with next stage.
iv) The state of the system at a stage is described by state variable.
v) When the current state is known, an optimal policy for the remaining stages is independent of the policy of the previous ones.
vi) The solution procedure begins by finding the optimal solution of each state from the optimal solutions of its previous stage.

**Steps of Dynamic Programming**

Dynamic programming design involves 4 major steps:

1. Develop a mathematical notation that can express any solution and sub-solution for the problem at hand.
2. Prove that the Principle of Optimality holds.
3. Develop a recurrence relation that relates a solution to its sub-solutions, using the mathematical notation of step 1. Indicates the initial values for that recurrence relation, and terms that signifies the final solution.
4. Write an algorithm to compute the recurrence relation.

## 4.4MULTISTAGE GRAPHS

A multistage graph G = (V, E)is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets $V_i$, $1 \leq i \leq k$. In addition, if <u,v> is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i \leq k$. The set $V_1$ 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 <i,j>. 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-costpath from s to t. Each set $V_i$ defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2, then to stage 3 and so on until it terminates at stage k. Fig 4.1 shows a five-stage graph. A minimum-cost path from s to t is indicated by the broken edges in the figure.

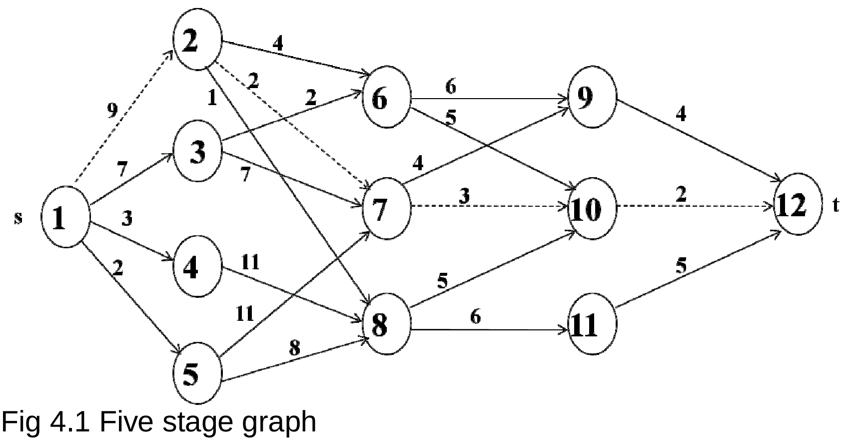



Fig 4.1 Five stage graph

A dynamic programming formulation for a k-stage graph problem is obtained by noticing the fact that, every path from s to tconsistof a sequence of k-2 decisions. The $i^{th}$ decision involves determining which vertex in $V_{i+1}$, $1 \leq i \leq k-2$, is to be on the path. It is easy to see that the principle of optimality holds for this problem. Let p(i, j) be a minimum cost path from vertex j in $V_i$ to vertex t. Let cost(i, j)be the cost of this path. Then using the forward formulation approach, we obtain:

$$\text{cost}(i, j) = \underset{\substack{l \in Vi+1 \\ <j,l> \in E}}{MIN} \{ c(j, l) + \text{cost}(i+1, l) \} \qquad \text{(Eq 4.1)}$$

Since cost(k-1, j ) = c(j, t), if <j,t>∈ E and cost(k-1, j) = α, if <j,t>∉E, the above equation may be solved for cost (1,s) by first computing cost(k-2, j) for all j∈$V_{k-2}$, then cost(k-3, j) for all j∈$V_{k-3}$, etc. and finally cost (1,s)

**Algorithm Graph_sortest_path (Graph G, k, n, p[ ])**

```
Step 1:  cost[n] = 0.0
Step 2:  for j= n-1 to 1
             //let r be a vertex such that <j,r> is an edge of G
             //and c[j][r] + cost [r] is minimum;
Step 3:     cost[j] = c[j][r] + cost [r]
Step 4:     d[j]=r;
Step 5:   end for
Step 6:   p[1]=1, p[k] =n
Step 7:   for j=2 to k-1
Step 8:      p[j]= d[p[j-1]]
Step 9:   end for
```

For the above algorithm we need to index the vertices of V from 1 to n. Indices are assigned according to stages. First index 1 is assigned to s, then the vertices in $V_2$ are indexed, then vertices in $V_3$, and so on, vertex t has index n.

The multistage graph problem can also be solved using the backwardapproach. Let bp(i,j) be a minimum-cost path from vertex s to a vertex jin $V_i$. Let bcost(i,j) be the cost of bp(i,j). From the backward approach weobtain

$$\text{bcost}(i, j) = \underset{\substack{l \in Vi-1 \\ <j,l> \in E}}{MIN} \{ c(j, l) + \text{bcost}(i-1, l) \} \qquad \text{(Eq 4.2)}$$

**Algorithm BGraph_sortest_path (Graph G, k, n, p[ ])**

```
Step 1:  bcost[1] = 0.0
Step 2:  for j= 2 to n
             //let r be a vertex such that <r, j> is an edge of G
             //and c[r][j] + bcost [r] is minimum;
Step 3:     bcost[j] = c[r][j] + bcost [r]
Step 4:     d[j]=r;
Step 5:   end for
Step 6:   p[1]=1, p[k] =n
Step 7:   for j=k-1 to 2
Step 8:      p[j]= d[p[j +1]]
Step 9:   end for
```

Since bcost(2,j) = c(1, j) if (1,j)∈E and bcoat(2,j) = ∞ if (1,j)∉ E,bcost(i,j) can be computed using (4.2) by first computing bcost for i = 3,then for i = 4, and so on.

## All-pairs shortest paths

Ifwewantto find the shortest pathnotjust betweensandt butbetweenall pairs of vertices then,oneapproachwouldbeto execute our generalshortest-path algorithmfrom |V| times, oncefor eachstartingnode. The totalrunningtime wouldthen beO($|V|^2|E|$). We'llnowseeabetteralternative, theO($|V|^3$)dynamicprogramming-basedFloyd-Warshall algorithm.

Finding a better algorithm by using dynamic programming approach, the first question came to our mind is that, whether a better sub-problem exists for computingdistancesbetweenallpairsofverticesin agraph? Simplysolvingtheproblemformoreandmorepairsorstartingpointsis unhelpful,becauseit leadsrightbacktotheO($|V|^2|E|$)algorithm.

Oneidea comesto mind isthat, the shortest pathu→ $w_1$→ …. →$w_l$→ vbetweenuandvusessomenumberof intermediate nodespossiblynone. Supposewedisallowintermediatenodesaltogether. Thenwecansolveall-pairsshortestpathsatonce,theshortestpathfromuto vis simplythe directedge(u, v), if it exists. Now let usgraduallyexpandthesetofpermissibleintermediate nodes. Wecandothisonenodeatatime,updatingtheshortestpathlengths ateachstage. Eventuallythissetgrowsto allofV, atwhichpointallthe verticesareallowedtobeonallpaths, andwehavefound thetrueshortestpathsbetweenverticesof thegraph.

Moreconcretely, numbertheverticesin Vas {1, 2,3 ….,n}, andlet dist(i;j;k)denotethelengthoftheshortestpathfrom itojinwhichonlynodes {1,2,…,k} canbeusedasintermediates. Initially,dist(i;j;0)isthelengthofthedirectedgebetween i andj, ifit exist, andis α otherwise.

Ifweexpandtheintermediatesettoinclude anextranodek,wemustreexamineallpairs i,jandcheckwhetherusingkasanintermediatepointgivesusashorterpathfrom i toj. Butthisis easy:ashortestpathfrom i tojthatuseskalongwithpossiblyother lower numberedintermediatenodesgoesthroughkis justonce. Andwehavealreadycalculatedthelength oftheshortestpathfromi tokandfromktojusingonly lowernumbered vertices.

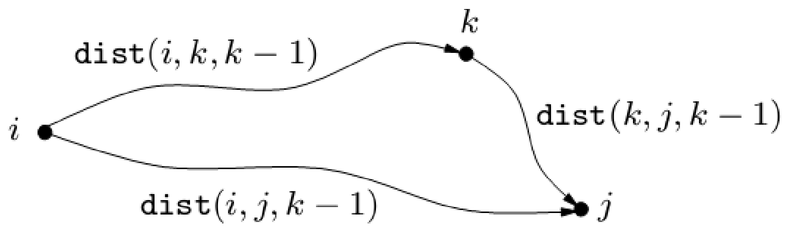



Fig 4.2 Computing Path

Thus, usingkgivesusashorterpathfromi tojifandonlyif
dist(i, k, k-1)+dist(k, j, k-1) <dist(i;j;k 1);
inwhichcasedist(i, j, k)shouldbeupdatedaccordingly.
HereistheFloyd-Warshallalgorithm – andit takesO(|V|$^3$)time.

**Algorithm All Path(cost, n)**

Step 1: for i := 1 to n
Step 2: for j :=1 to n
Step 2: dist(I,j,0) := 1;
Step 3: end for
Step 5: end for
Step 4: for all (i, j) ∈ E
Step 5: dist (i, j, 0) = ℓ (i, j)
Step 6: end for
Step 7: for k:=1 to n
Step 8: for i:=1 to n
Step 7: for j :=1 to n
Step 8: dist(i, j,k) = min {dist(i,k,k-1) + dist(k, j, k-1), dist(i,j,k-1)}
Step 9: end for
Step 11: end for
Step 12: end for

## Single Source Shortest Path

**Problem:** Given a directed graph *G(V,E)* with *weighted edgesw(u,v)*, define the *path weight* of a path *p* as

$$w(p)=\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

For a given source vertex *s*, find the *minimum weight paths* to every vertex reachable from *s* denoted

$$\delta(s,v)=\begin{cases} \min\{w(p) \quad s \to v\} \\ \infty \text{ otherwise} \end{cases}$$

The final solution will satisfy certain caveats:

- The graph cannot contain any *negative weight cycles* (otherwise there would be no minimum path since we could simply continue to follow the negative weight cycle producing a path weight of -∞).
- The solution cannot have any *positive weight cycles*
- The solution can be assumed to have no zero weight cycles (since they would not affect the minimum value).

Therefore given these caveats, we know that the shortest paths must be *acyclic* (with ≤ |*V*| distinct vertices) ⇒ ≤ |*V*| - 1 edges in each path.

We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph.

Let $dist^{l}[u]$ 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, $dist^{1}[u] = cost[v,u]$, $1 \le u \le n$. 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.

Our goal then is to compute $dist^{n-1}[u]$ for all u. This can be done using the dynamic programming methodology. First, we make the following observations:

1. If the shortest path from v to u with at most k, k > 1, edges has not 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 (j,u). The path from v to j has k - 1 edges, and its length is $dist^{k-1}[j]$. All vertices j such that the edge (j, u) is in the graph are candidates for j. 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.

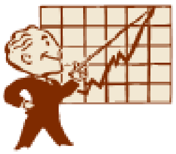
These observations result in the following recurrence for dist:

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

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

**Algorithm BellmanFord(v, cost, dist, n)**

```
Step 1:  for i := 1 to n do
Step 2:      dist[i] := cost[v, i];
Step 3:  end for
Step 4:  for k := 2 to n - 1 do
Step 5:      for each u such that u≠v and u has at least
             one incoming edge
Step 6:          for each <i, u> in the graph
Step 7:              if dist[u] >dist[i] + cost[i,u]
Step 8:                  dist[u] := dist[i] + cost[i,u];
Step 9:              end if
Step 10:         end for
Step 11:     end for
Step 12: end for
```



**CHECK YOUR PROGRESS**

1. State True or False

a) All the problems can be solved by using dynamic programming technique.
b) To solve a problem by using dynamic programming, the problem must have to possess principle of optimality.
c) A multistage graph can have a cycle.
d) A optimal binary search tree is a binary search tree which has minimal expected cost of locating each node

## 4.6 0/1 KNAPSACK PROBLEM USING DYNAMIC PROGRAMMING

In the previous unit, we have discussed about the Knapsack problem, and found that fractional knapsack problem can be solved by using greedy strategy. The 0-1 knapsack problem can only be solved by using dynamic programming. Below we will discuss methods for solving 0-1 knapsack problem.

The naive way to solve this problem is to cycle through all $2^n$ subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack. But, we can find a dynamic programming algorithm that will usually do better than this brute force technique.

Our first attempt might be to characterize a sub-problem as follows:

Let $S_k$ be the optimal subset of elements from $\{I_0, I_1,...I_k\}$. But what we find is that the optimal subset from the elements $\{I_0, I_1,... I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1,...I_k\}$ in any regular pattern. Basically, the solution to the optimization problem for $S_{k+1}$ might NOT contain the optimal solution from problem $S_k$.

To illustrate this, consider the following example:

| Item | Weight | Value |
|---|---|---|
| $I_0$ | 3 | 10 |
| $I_1$ | 8 | 4 |
| $I_2$ | 9 | 9 |
| $I_3$ | 8 | 11 |

The maximum weight the knapsack can hold is 20.

The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$ but the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$. In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$. Instead it builds upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12.

So, now, let us rework on our examplewith the following idea:Let B[k, w] represents the maximum total value of a subset $S_k$ with weight w. Our goal is to find B[n, W], where n is the total number of items and W is the maximal weight, the knapsack can carry.

Using this definition, we have $B[0, w] = w_0$, if $w \geq w_0$.
$= 0$, otherwise

Now, we can derive the following relationship that B[k, w] obeys:

$B[k, w] = B[k - 1,w]$, if $w_k > w$
$= \max \{ B[k - 1,w], B[k - 1,w - w_k] + v_k\}$

In general:

1) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ...I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight w, if weights of item k is greater than W.
Basically, we can NOT increase the value of our knapsack with weight w if the new item we are considering weighs more than W – because it WON'T fit!!!

2) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_1, I_2, ... I_{k-1}\}$ with weight w, if item k should not be added into the knapsack.

3) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, ... I_{k-1}\}$ with weight $w$-$w_k$, plus item k.

You need to compare the values of knapsacks in both case 2 and 3 and take the maximal one.

Recursively, we will still have an $O(2^n)$ algorithm. But, using dynamic programming, we simply perform in just two loops - one loop running n times and the other loop running W times.

Here is a dynamic programming algorithm to solve the 0/1 Knapsack problem:

Input: S, a set of n items as described earlier, W the total weight of the knapsack. (Assume that the weights and values are stored in separate arrays named w and v, respectively.)

Output: The maximal value of items in a valid knapsack.

```
int i, k;
for (i=0; i<= W; i++)
    B[i] = 0

for (k=0; k<n; k++)
{
for (i = W; i>= w[k]; i--)
        {
        if (B[i – w[k]] + v[k]> B[i])
            B[i] = B[i – w[k]] + v[k]
        }
}
```

Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops. When comparing this with the previous $O(2^n)$, we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient.

Let's run through an example:

| I | Item | $w_i$ | $v_i$ |
|---|---|---|---|
| 0 | $I_0$ | 4 | 6 |
| 1 | $I_1$ | 2 | 4 |
| 2 | $I_2$ | 3 | 5 |
| 3 | $I_3$ | 1 | 3 |
| 4 | $I_4$ | 6 | 9 |
| 5 | $I_5$ | 4 | 7 |

W = 10

| Item | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 6 | 6 | 6 | 6 | 6 | 6 | 6 |
| 1 | 0 | 0 | 4 | 4 | 6 | 6 | 10 | 10 | 10 | 10 | 10 |
| 2 | 0 | 0 | 4 | 5 | 6 | 9 | 10 | 11 | 11 | 15 | 15 |
| 3 | 0 | 3 | 4 | 7 | 8 | 9 | 12 | 13 | 14 | 15 | 18 |
| 4 | 0 | 3 | 4 | 7 | 8 | 9 | 12 | 13 | 14 | 16 | 18 |
| 5 | 0 | 3 | 4 | 7 | 8 | 10 | 12 | 14 | 15 | 16 | 19 |

## 4.7 TRAVELLING SALESMAN PROBLEM

We have seen how to apply dynamic programming to a subset selectionproblem (0/1 knapsack). Now we turn our attention to a permutation problem.Note that permutation problems usually are much harder to solve thansubset problems as there are n! different permutations of n objects whereasthere are only $2^n$different subsets of n objects (n!> 2n). Let G = (V,E)be a directed graph with edge costs $c_{ij}$. The variable $c_{ij}$is defined such that$c_{ij}$> 0 for all i and j and $c_{ij} = \alpha$ if (i,j) $\notin$ E. Let |V| = n and assumen > 1. A tour of G is a directed simple cycle that includes every vertex inV. The cost of a tour is the sum of the cost of the edges on the tour. Thetraveling salesperson problem is to find a tour of minimum cost.

Different problems can be viewed as the traveling salesman problem.For example, suppose we have to define the route a postal van to pick up mail from mail boxes located at n different sites. If we represent the situation by graphs then the vertices of the graph will be different cities and the edges of the graph are the paths between two cities and the weight of a edge can be the distance between the cities. Our task is to find the route taken by the postal van is a tour with minimum cost or length.

In the following discussion, without losing the main concept, we takethe tour as a simple path that starts and ends at thestarting vertex. 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 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 in V - {1,k}. Hence, the principle of optimality holds. 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(1, V - {1}) is the length of an optimal salesman’s tour. From the principal of optimality it follows that

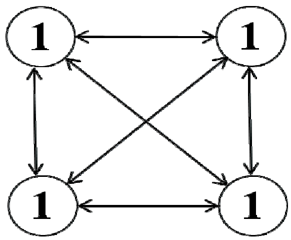
$$g(1, V-\{1\}) = \min_{2\le k\le n} \{c_{1k} + g(k, V - \{1, k\})\}$$

In general

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

The above equation can be solved for g(1, V - {1}) if we know g(k, V - {1, k}) for all choices of k. The g values can be obtained by using this equation. Clearly, g(i, Ø) = $c_{j1}$, 1≤ i ≤ n. Hence, we can use this equation to obtain g(i, S) for all S of size 1. Then we can obtain g(i,S) for S with |S| = 2, and so on. When |S|< n - 1, the values of i and S for which g(i, S) is needed are such that i ≠ 1, 1$\notin$ S, and i $\notin$S.

Consider the directed graph of Fig 4.5(a). The edge lengths are given by matrix c of Fig 4.5(b)




**(a)**

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

**(b)**

Fig 4.5: Directed graph and Edge matrix c

Thus,

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

Using the above equation we obtain

| | |
|---|---|
| $g(2,\{3\}) = c_{23} + g(3,\varnothing) = 15$ | $g(2,\{4\}) = 18$ |
| $g(3,\{2\}) = 18$ | $g(3,\{4\}) = 20$ |
| $g(4,\{2\}) = 13$ | $g(4,\{3\}) = 15$ |

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

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

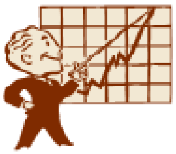
Finally, we obtain

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

An optimal tour of the graph of Figure has length 35. A tour of this length can be constructed if we retain with each g(i, S) the value of j that minimizes the right-hand side of the graph. Let J(i,S) be this value. Then, J(1,{2,3,4}) = 2. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from g(2, {3, 4}). SoJ(2, {3, 4}) = 4. Thus the next edge is (2,4). The remaining tour is for g(4, {3}). So J(4, {3}) = 3. The optimal tour is 1, 2, 4, 3, 1.

hasn't yet beenscheduled, schedule job j at the rightmost available spot. If j hasalready been scheduled, go to the next number in the sequence.

Note that the above rule also correctly positions jobs with $a_i = 0$. Hence,these jobs need not be considered separately.



**CHECK YOUR PROGRESS**

2. Sate True or False.

a) 0/1 knapsack problem can also be solved by using greedy strategy.

b) Travelling Salesman problem is to find out the shortest cycle in the graph covering all the vertices

c) 0/1 knapsack does not possess a optimal substructure.

d) Flow shop scheduling problem is to find out the optimal sequence to run n jobs in m processors.

## 4.9 LET US SUM UP

- A problem can be solved by dynamic programming only when it possesses optimal substructure.
- A problem is said to satisfy the principle of optimality, if the sub solutions of an optimal solution of the problem are themselves optimal solution for their sub problems.
- In dynamic programming wefirst solve the sub-problems and then use these solutions to get the optimal solution in recursive manner.

## 4.10 FURTHER READINGS

1. T. H. Cormen, C. E. Leiserson, R.L.Rivest, and C. Stein, "Introduction to Algorithms", Second Edition, Prentice Hall of India Pvt. Ltd, 2006.

2. Ellis Horowitz, SartajSahni and SanguthevarRajasekaran, Fundamental of data structure in C, Second Edition, Universities Press, 2009.
3. Alfred V. Aho,John E. Hopcroft and Jeffrey D. Ullman, "The Design and Analysis of Computer Algorithms", Pearson Education, 1999.
4. Ellis Horowitz, SartajSahni and SanguthevarRajasekaran, Computer Algorithms/ C++, Second Edition, Universities Press, 2007.

.

## 4.11 ANSWERS TO CHECK YOUR PROGRESS

1. a) Falseb)Truec) Falsed) True

1. a) Falseb) Truec) Falsed) True

## 4.12 PROBABLE QUESTIONS

1. Explain the characteristics of dynamic programming.
2. Describe the steps of dynamic programming algorithm.
3. Solve 0/1 knapsack problem using dynamic programming.
4. Flow shop scheduling algorithm possess the optimal sub-structure, explain it.
5. With an example explain how 0/1 knapsack problem can be solved by using dynamic programming.
6. Describe the method of solving travelling salesman problem using dynamic programming.
7. Explain, what optimal binary search tree is.