UNIT - III

DYNAMIC PROGRAMMING

3.1 INTRODUCTION

- Dynamic programming has evolved into a major paradigm of algorithm design in computer science. However, its name is something of a mystery to many people. The name was coined in 1957 by Richard Bellman to describe a type of optimum control problem. Actually, the name originally described the problem more than the technique of solution. The sense in which programming is meant is "a series of choices". Like the programming of the radio station. The word dynamic conveys the idea that choices may depend on the current state, rather than being decided ahead of time. A radio show in which listeners phone in their requests might be said to be "dynamically programmed" in contrast with the usual format where the selections of songs are decided before the show begins. Bellman described a method of solution for "dynamic programming" problems, which has become the inspiration for many computer algorithms. The main feature of this method was that it replaced an exponential time computation by a polynomial time computation. That continues to be a common feature of dynamic programming algorithms.
- In algorithms we have studied so far, correctness tended to be easier than efficiency. In optimization problems, we are interested in finding the solution that maximizes or minimizes the same function. In designing algorithms for optimization problem, we must design algorithm that gives the best possible solution.
- Greedy algorithms, which take the best local decision of each step, occasionally produce a global optimum solution, but we need to prove the same.
- Dynamic programming is a technique for computing recurrence relations efficiently by sorting partial results.
- A dynamic programming algorithm stores results, or solutions, for small sub-problems. Later on it uses these stored solutions instead of recomputing them, to solve larger subproblems. Thus, dynamic programming is especially well suited to problems where a recursive algorithm would solve many of the subproblems repeatedly.
- We will introduce a characterization of dynamic programming algorithms that provides a unified framework for a wide variety of published algorithms that might seem quite different on the surface. This framework permits a recursive solution to be converted into a dynamic programming algorithm and provides a way to analyze the complexity of the dynamic programming algorithm.

3.2 THE GENERAL METHOD

- Dynamic programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.
- Like greedy method; for many problems, it is not possible to make stepwise decisions (based only on local information) in such a manner that the sequence of decisions made is optimal.
- One way to solve such problems is to try out all possible decision sequences. We could enumerate all decision sequences and then pick out the best. Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal.
- In dynamic programming, an optimal sequence of decisions is arrived at by making explicit appeal to the principle of optimality.

3.3 FOUR STEPS TO DEVELOP A DYNAMIC PROGRAMMING ALGORITHM

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution, which is the most creative work.
3. Compute the value of an optimal solution in a bottom-up fashion. We can also use recursive method.
4. Construct an optimal solution from computed information by making use of computed results.

Generic Problem Structure :

$$t_n = \begin{cases} \text{constant value} & \text{,if trivial (p)} \\ \text{combine f(p1), f(p2), ... f(pn)} & \text{, otherwise} \end{cases}$$

3.4 ELEMENTS OF DYNAMIC PROGRAMMING

- Following are the three elements of dynamic programming :
  - ➢ Optimal substructure
  - ➢ Overlapping subproblems
  - ➢ Memorization

A Dynamic Programming Solution has Three Components :

1. Formulate the answer as a recurrence relation or recursive algorithm.
2. Show that the number of different instances of your recurrence is bounded by a polynomial.
3. Specify an order of evaluation for the recurrence.

- To decide whether a problem can be solved using dynamic programming method, the following elements o programming should be considered :
  - ➢ Optimal substructure
  - ➢ Overlapping sub-problems.
  - ➢ Memorization

Optimal Substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal so subproblems. If a problem exhibits optimal substructure, then it means that dynamic programming (ar method) might apply. As the optimal solution for the problem is built from the optimal solution of the subpr is necessary to consider those subproblems which have an optimal solution.
- The execution time of a dynamic programming algorithm depends on the product of two factors : the overa of subproblems and how many choices we look at for each subproblem.
- Dynamic programming uses optimal substructure in a bottom-up fashion. It first finds optimal sol subproblems. When the subproblems are solved, then it finds an optimal solution to the problem.

Overlapping Subproblems

- When a recursive algorithm revisits the same problem over and over again, then it is said that the opt problem has overlapping subproblems. This is beneficial for dynamic programming. It solves each subprob and stores the answer in a table. This answer can be searched in constant time when required.
- This is contradictory to divide-an-conquer strategy where a new problem is generated at each step of recursio similar to the recursive algorithm.

Memorization

- Generally dynamic programming maintains a table for solutions of sub-problems. But it uses the control
- In a memorized recursive algorithm, an entry is maintained in a table for solution of each subproblem. In entries contain a special value which indicates that entry is not yet used. For each subproblem which is enco for the first time, its solution is computed and stored in the table. Next time for that subproblem, its entry is s and value is used. This can be implemented using hashing.

3.5 GENERIC DYNAMIC PROGRAMMING

Algorithm: Generic Dynamic Programming

```
function solve(p)
begin
  if known (p) then
    return (lookup (p))
```

```
else
   x= compute (p)
   save(p, x)
   return x
end

function compute(p)
begin
  if trivial(p)
    then return(trivial_sol(p))
  else
    divide p into subproblems p1, p2, ...pn
    S1 = solve(p1), .....
    Sn = solve (pn)
    return (combine(S1, S2, ..., Sn))
End
```

solve + compute	Recursive definitions
lookup	Maintains a table to record (Problem p, solution-for-p)
trivial	Gives a solution (always a constant value) for p
trivial_sol	Judges whether we can use trivial to compute p
divide	It depends on the problem structure
combine	It depends on the problem structure

3.6 WHEN CAN WE USE DYNAMIC PROGRAMMING ?

- Dynamic programming computes recurrences efficiently by storing partial results. Thus dynamic programming can only be efficient when there are not too many partial results to compute.
- There are n! Permutations of an n-element set - we cannot use dynamic programming to store the best solution for each sub-permutation. There are 2^n subsets of an n-element set.
- However, there are only n(n – 1)/2 contiguous substrings of a string, each described by a starting and ending point, so we can use it for string problems.
- There are only n(n – 1)/2 possible subtrees of a binary search tree, each described by a maximum and minimum key, so we can use it for optimizing binary search trees.
- Dynamic programming works best on objects which are linearly ordered and cannot be rearranged. For example, characters in a string, matrices in a chain, points around the boundary of a polygon, the left-to-right order of leaves in a search tree.
- Whenever your objects are ordered in a left-to-right way, you should smell dynamic programming.

3.7 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 the Greedy Method and the Dynamic Programming:** The essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequences is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal if the principle of optimality holds and so will not be generated. One may

feel that in this method, one has to look at all possible decision sequences to obtain an optimal decision se using dynamic programming. This is not the case. Because of the use of the principle of optimality, decision seq containing subsequences that are suboptimal are not considered. Although the total number of different de sequences is exponential in the number of decisions, dynamic programming algorithms often have a polyn complexity. The exponential number of decisions can be generated because if there are d choices for each de decisions to be made then there are d^n possible decision sequences.

- Another important feature of the dynamic programming approach is that optimal solutions to subproble retained so as to avoid recomputing their values. The use of these tabulated values makes it natural to reca recursive equations into an iterative algorithm. Most of the dynamic programming algorithms are expressed o this way.
- To illustrate the difference between the two techniques, let us see two variants of a classical optimization proble
- The **0-1 Knapsack Problem** is posed as follows. A thief robbing a store finds n items; the i^{th} item is worth v_i and weighs ω_i pounds, where v_i and ω_i are integers. He wants to take as valuable a load as possible, but he can c most W pounds in his knapsack for some integer W. Which items should he take ? (This is called the 0-1 kn problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an take an item more than once.)
- In the **Fractional Knapsack Problem**, the setup is the same, but the thief can take fractions of items, rath having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as be a gold ingot, while an item in the fractional knapsack problem is more like gold dust.
- For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j fr load, the remaining load must be the most valuable load weighing at most $W - \omega_j$ that the thief can take fr $n - 1$ original items excluding j. For the comparable fractional problem, consider that if we remove a weight ω item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - \omega$ that t can take from the $n - 1$ original items plus $\omega_j - \omega$ pounds of item j.
- Although the problems are similar, the fractional knapsack problem is solvable by a greedy strategy, whereas problem is not. To solve the fractional problem, we first compute the value per pound v_i/ω_i for each item. Ob greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with t greatest value per pound, and so forth until he can't carry any more. Thus, by sorting the items by value per the greedy algorithm runs in O(n log n) time.

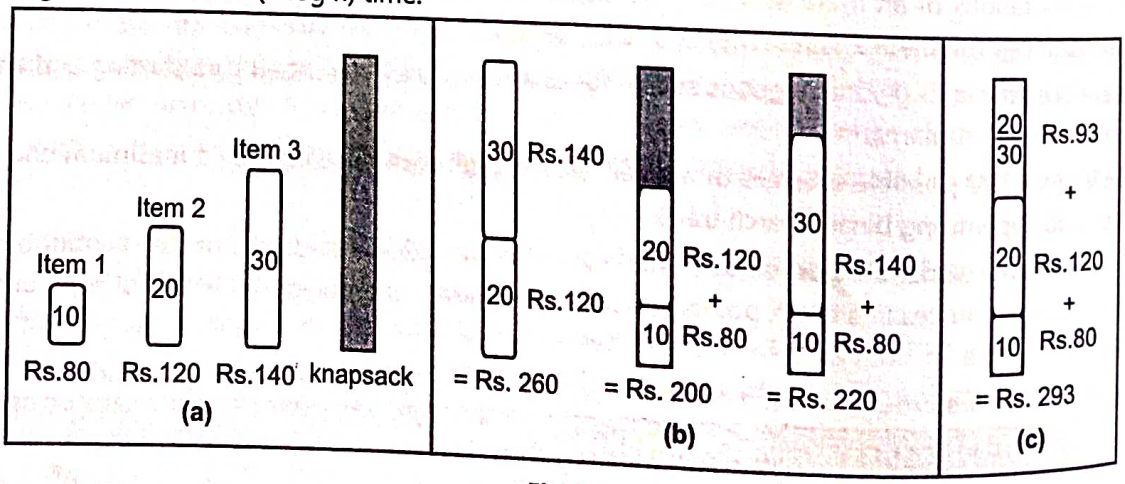


Fig. 3.1

- The greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack prob taking the items in order of greatest value per pound yields an optimal solution.
- To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illust in Fig. 3.1 (a). There are 3 items, and the knapsack can hold 50 pounds Item 1 weighs 10 pounds and is worth

Item 2 weighs 20 pounds and is worth Rs. 120. Items 3 weighs 30 pounds and is worth Rs. 140. Thus, the value per pound of item 1 is Rs. 8 per pound which is greater than the value per pound of either item 2 (Rs. 6 per pound) or item 3 (Rs. 4.67 per pound). The greedy strategy, therefore, would take item 1 first. As can be seen from the case analysis in Fig. 3.1 (b), however, the optimal solution takes items 2 and 3, leaving 1 behind. The two possible solutions that involve item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Fig. 3.1 (c). Taking item 1 does not work in the 0-1 problem because the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider an item for inclusion in the knapsack, we must compare the solution to the subproblem in which the item is included with the solution. Thus, dynamic programming can be used to solve the 0-1 problem which gives better solution than greedy method.

8 PECULIAR CHARACTERISTICS AND USE OF DYNAMIC PROGRAMMING

Solution to a problem is viewed as a result of a sequence of decisions.

Avoids enumeration of some decision sequences that cannot be possibly optimal.

An optimal sequence of decisions is arrived at by making an explicit appeal to the 'principle of optimality'.

Principle of optimality states that : An optimal sequence of decisions has 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 the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated.

However, sequences containing suboptimal sequences cannot be optimal and so will not be generated.

Two approaches for dynamic programming: Let $(x_1, x_2, ..., x_n)$ be variables.

- **Forward Approach:** Decision x_i is made in terms of optimal decision sequences for $x_{i+1}, ..., x_n$.
- **Backward Approach:** Decision x_i is made in terms of optimal decision sequences for $x_1, x_2, ..., x_{i-1}$.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type.

Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller sub-problem only once and recording the results in a table from which we can obtain a solution to the original problem.

Applicability of dynamic programming to an optimization problem requires the problem to satisfy the principle of optimality : an optimal solution to any of its instances must be made-up of optimal solutions to its sub-instances.

9 LIMITATIONS OF DYNAMIC PROGRAMMING

Dynamic programming can be applied to any problem that observes the *principle of optimality*. Roughly stated, this means that partial solutions can be optimally extended with regard to the *state* after the partial solution instead of the partial solution itself. For example, to decide whether to extend an approximate string matching by a substitution, insertion, or deletion, we do not need to know exactly which sequence of operations was performed to date. In fact, there may be several different edit sequences that achieve a cost of C on the first p characters of pattern P and t characters of string T. Future decisions will be made based on the *consequences* of previous decisions, not the actual decisions themselves.

- Problems do not satisfy the principle of optimality if the actual operations matter, as opposed to just the cost of the operations. Consider a form of edit distance where we are not allowed to use combinations of operations in certain particular orders. Properly formulated, however, most combinatorial problems respect the principle of optimality.
- The biggest limitation on using dynamic programming is the number of partial solutions we must keep track of. For all of the examples we will see, the partial solutions can be completely described by specifying the stopping places in the input. This is because the combinatorial objects being worked on (strings, numerical sequences, and polygons) all have

an implicit order defined upon their elements. This order cannot be scrambled without completely changing the problem. Once the order is fixed, there are relatively few possible stopping places or states, so we get efficient algorithms. If the objects are not firmly ordered, however, we have an exponential number of possible partial solutions and are doomed to need an infeasible amount of memory.

3.10 OPTIMAL BINARY SEARCH TREE (OBST)

- For the same set of keys, we can draw a binary search tree in different ways.
- For example, let us construct different binary search trees for keys = {Santosh, Anand, Amol} Different binary search trees are as shown in Fig. 3.2.

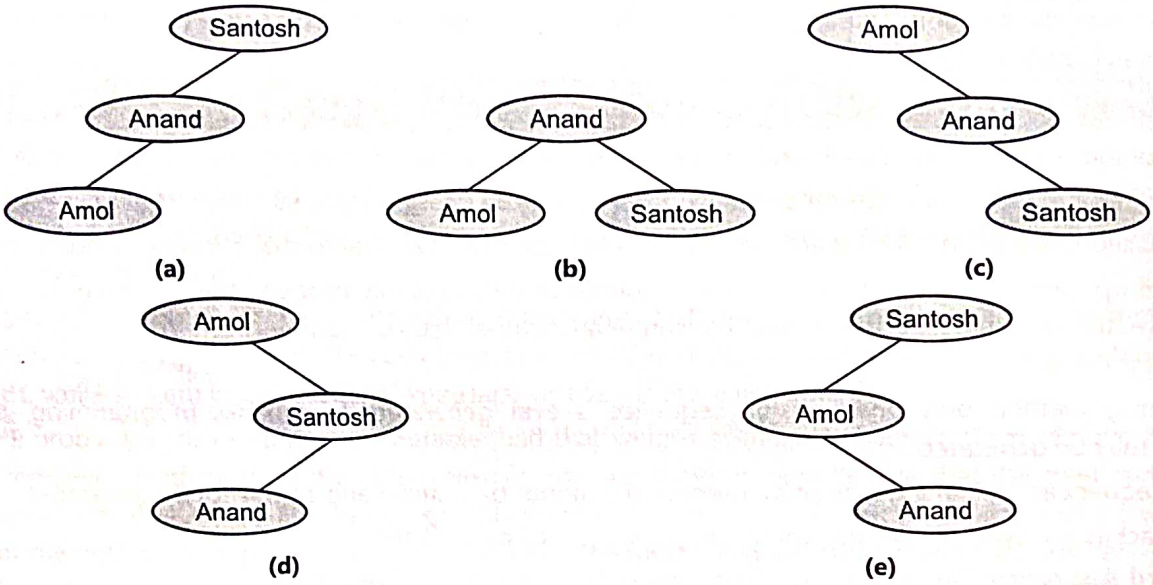


Fig. 3.2

- So for the given keys, there are total 5 binary search trees possible. But we are always interested in binary search tree having minimum search time. In the above figures, Fig. 3.2 (b) shows an optimal binary search tree, because it requires only 2, 1, 2 comparisons for searching keys {Santosh, Anand, Amol} respectively.
- Consider the keys $\{k_1, k_2, ..., k_n\}$ such that $k_1 < k_2 < k_3 < ... < k_n$. Let every successful search for key k_i probability p(i). Also every unsuccessful search for key x has probability of failure q(i) for $0 \leq i \leq n$, and $k_i < x < k_{i+1}$. We can add fictitious node as a child for every leaf node. So the 2 resulting BSTs for keys = {Aditya, Pratik, Sneha} are shown below in Fig. 3.3.

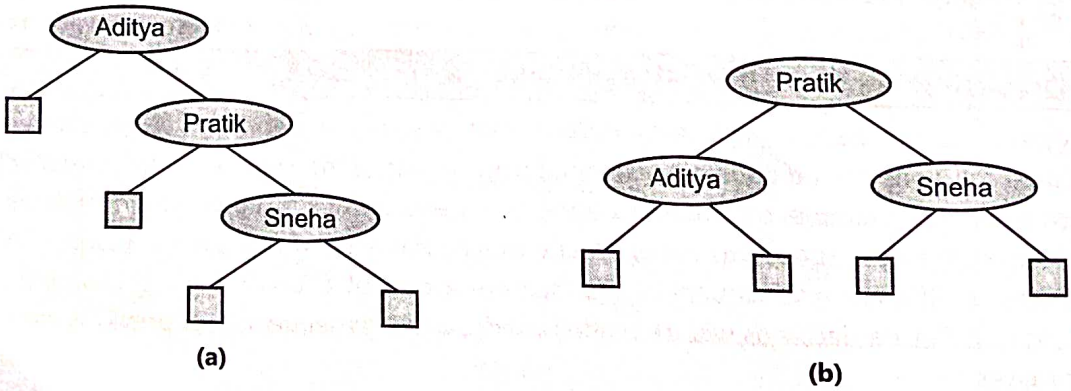


Fig. 3.3

- Here all the keys represent internal nodes. All successful searches will always end at an internal node. All square denote external nodes which are fictitious. All unsuccessful searches will end at some external node. If there are n keys there are n + 1 external nodes. So all the keys which are not part of BST belong to one of n + 1 equivalence class E_i for $0 \leq i \leq n$. Class E_0 contains all keys $m < k_1$. Class E_1 contains all keys $m > k_1$ but $m < k_2$. In general class E_i contains all keys $m > k_i$ but $m < k_{i+1}$. So if an unsuccessful search reaches at node E_i at level l, it means that $l - 1$ comparisons a

already performed. Hence cost of such node is q(i) * (level (E_i) – 1). Similarly every successful search which stops at key k_i at level *l* has cost p(i) * level (k_i).

Hence cost of binary search tree is :

$$\sum_{1 \le i \le n} p(i) * \text{level}(k_i) + \sum_{0 \le i \le n} a(i) * (\text{level}(E_i) - 1)$$

Now we can define optimal BST.

Definition : An **optimal BST** is a BST with minimum cost.

Let us take an example.

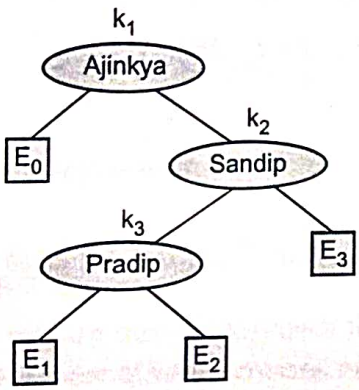
SOLVED EXAMPLES

Example 3.1 : *Consider the keys = {Ajinkya, Sandip, Pradip}. There are total 5 BSTs possible.*

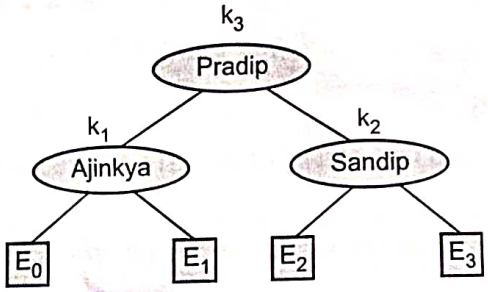
Let probabilities p(i) = q(i) = 2 for all i.

Calculate cost of above trees. [For convenience, values of probabilities are multiplied by a constant value.]

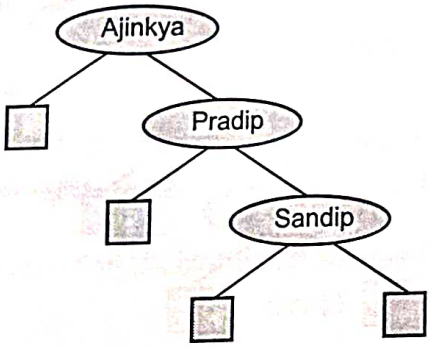
Solution:



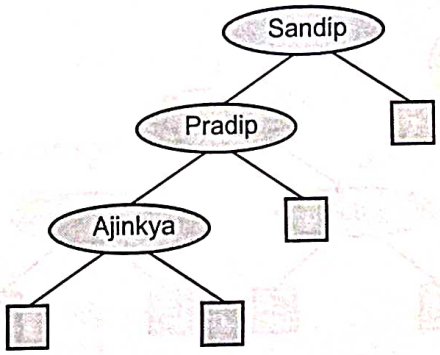
(a)



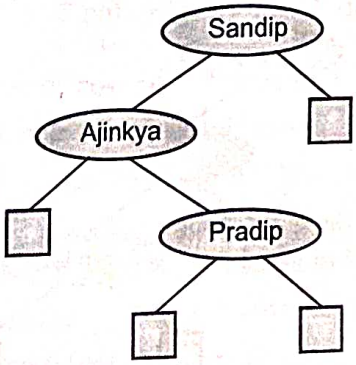
(b)



(c)



(d)



(e)

Fig. 3.4

For Fig. 3.4 (a) :

$$\text{Cost} = \sum_{1 \le i \le 3} p(i) * \text{level}(k_i) + \sum_{1 \le i \le 3} q(i) * (\text{level}(E_i) - 1)$$

$$\sum_{1 \le i \le 3} p(i) * \text{level}(k_i) = p_1 * \text{level}(k_1) + p_2 * \text{level}(k_2) + p_3 * \text{level}(k_3)$$

$$= 2 * 1 + 2 * 2 + 2 * 3 = 2 + 4 + 6 = 12$$

$$\sum_{1 \le i \le 3} q(i) * (\text{level}(E_i) - 1) = q_0 * (\text{level}(E_0) - 1) + q_1 * (\text{level}(E_1) - 1) + q_2 * (\text{level}(E_2) - 1)$$

$$+ q_3 * (\text{level}(E_3) - 1)$$

$$= 2 * 1 + 2 * 3 + 2 * 3 + 2 * 2$$

$$= 2 + 6 + 6 + 4 = 18$$

For Fig. 3.4 (a), cost = 12 + 18 = 30

For Fig. 3.4 (b) :

$$\sum_{1 \le i \le 3} p(i) * \text{level}(k_i) = 2 * 2 + 2 * 2 + 2 + 2 * 1$$

$$= 4 + 4 + 2 = 10$$

$$\sum_{1 \le i \le 3} q(i) * \text{level}(k_i) = 2 * 2 + 2 * 2 + 2 * 2 + 2 * 2 = 16$$

For Fig. 3.4 (b), cost = 10 + 16 = 26

Like this, you can find cost of other BSTs for the same keys. But Fig. 3.4 (b) has smallest cost, hence it is an optimal BST for the given keys.

Example 3.2 : *Consider the keys = {while, do, if} and p(i) = q(i) = 1/7 for all i. Find the cost of all possible BSTs and find OBST.*

Solution :

The 5 possible BSTs for the given keys are shown below in Fig. 3.5.

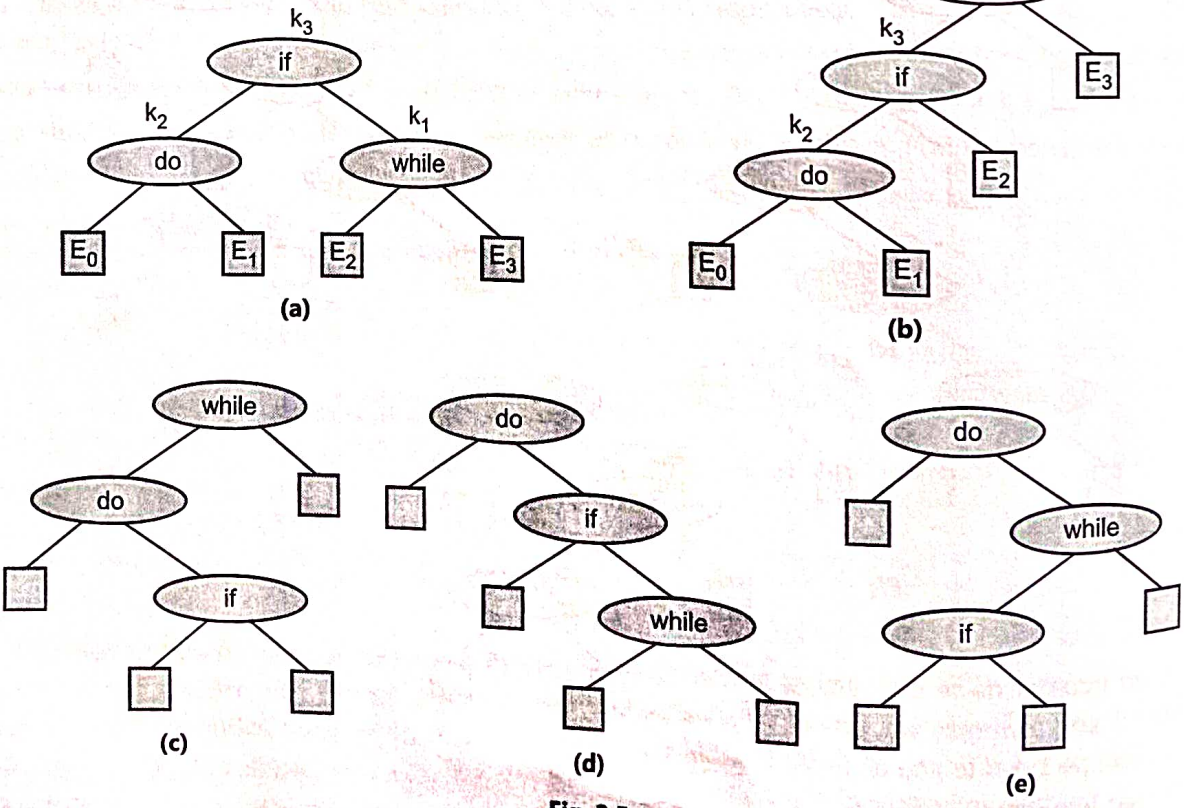


Fig. 3.5

For Fig. 3.5 (a) :

$$\text{cost} = \sum_{1 \le i \le n} p(i) \text{ level } (k_i) + \sum_{1 \le i \le n} q(i) \sum q(i) * (\text{level}(E_i) - 1)$$

$$\sum_{1 \le i \le n} q(i) \text{ level } (k_i) = p_1 * \text{level } (k_1) + p_2 * \text{level } (k_2) + p_3 * \text{level } (k_3)$$

$$= 1/7 \ (2 + 2 + 1)$$

$$= 5/7$$

$$\sum_{1 \le i \le n} q(i) * (\text{level}(E_i) - 1) = q_0 * (\text{level } (E_0) - 1) + q_1 * (\text{level } (E_1) - 1) + q_2 * (\text{level } (E_2) - 1)$$

$$+ q_3 * (\text{level } (E_3) - 1)$$

$$= 1/7 \ (2 + 2 + 2 + 2)$$

$$= 8/7$$

For Fig. 3.5 (a), $\text{cost} = (5/7) + (8/7) = 13/7$

For Fig. 3.5 (b) :

$$\sum_{1 \le i \le n} p(i) \sum p(i) * \text{level } (k_i) = 1/7 \ (1 + 3 + 2)$$

$$= 6/7$$

$$\sum_{1 \le i \le n} q(i) * (\text{level } (E_i) - 1) = 1/7 \ (3 + 3 + 2 + 1)$$

$$= 9/7$$

For Fig. 3.5 (b), $\text{cost} = (6/7) + (9/7) = 15/7$

Similarly, for Fig. 3.5 (c), Fig. 3.5 (d) and Fig. 3.5 (e), cost = 15/7.

So Fig. 3.5 (a) is OBST for given keys.

Practically we can not use such an approach to find OBST, because we have to draw all possible BSTs and then find cost of all BSTs. As number of keys increases, number of BSTs also increases.

Dynamic programming approach can be used to construct an OBST stepwise where the principle of optimality should hold at each step.

ssume that there are n keys $\{k_1, k_2, ..., k_n\}$ where $k_1 < k_2 < ... < k_n$. So at some step, if is a root of tree, then

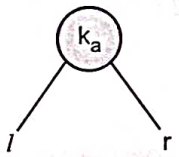


Fig. 3.6

Because this is a BST, the left subtree *l* has keys $k_1, k_2, ..., k_{a-1}$ and external nodes $E_1, ..., E_{a-1}$. Also the right subtree r has keys $k_{a+1}, ..., k_n$ and external nodes $E_a, ..., E_n$.

Therefore using equation (1), cost of left subtree *l* is

$$\text{cost } (l) = \sum_{1 \le i \le a} p(i) * \text{level } (k_i) + \sum_{1 \le i \le a} q(i) * (\text{level } (E_i) - 1) \quad \text{... (2)}$$

Similarly, cost of right subtree r using equation (1) is

$$\text{cost } (r) = \sum_{1 \le i \le n} p(i) * \text{level } (k_i) + \sum_{1 \le i \le n} q(i) * (\text{level } (E_i) - 1) \quad \text{... (3)}$$

Therefore cost of tree in Fig. 3.6 is addition of cost of left subtree *l*, cost of right subtree r, probability of node k_a, eight of nodes 0 to a – 1, weight of nodes a to n. In notation, this can be stated as :

$$p(a) + \text{cost } (l) + \text{cost } (r) + w(0, a - 1) + w(a, n) \quad \text{... (4)}$$

Cost (*l*) and cost (r) are determined considering their roots at level 1. If cost (*l*) is minimum and cost (r) is also minimum, then cost of equation (4) is also minimum, and therefore tree in Fig. 3.5 (a) is optimal.

Let c(i, j) denotes cost of OBST t_{ij} having keys $k_{i+1}, ..., k_j$ and external nodes $E_i, ..., E_j$. So for left subtree *l* of OBST in Fig. 3.6, cost (l) = c(0, a-1) and for its right subtree r, cost (r) = c(a, n).

Hence equation (4) becomes

$$p(a) + c(0, a - 1) + c(a, n) + w(0, a - 1) + w(a, n) \quad \text{... (5)}$$

Equation (5) gives cost of a tree having nodes from k_0 to k_n. In general, we can write an equation which gives cost for a subtree having nodes from k_i to k_j, as

$$p(a) + c(i, a-1) + c(a, j) + w(i, a-1) + w(a, j) \quad \ldots (6)$$

Obviously equation (6) gives minimum cost only if a is chosen properly. So we have to solve equation (6) for different values of a and then select minimum. Hence we can generalize equation (6) to get

$$c(i, j) = \min [c(i, a-1) + c(a, j) + w(i, a-1) + p(a) + w(a, j)]$$

which can be simplified as :

$$c(i, j) = \min_{i \le a \le j} [c(i, a-1) + c(a, j)] + w(i, j)] \quad \ldots (7)$$

Here are the steps to find OBST :

- First c(i, i) = 0, r(i, i) = 0 and w(i, i) = q(i) for $0 \le i \le n$, where n is the number of keys.
- Then compute c(i, j) for j – i = 1. Also compute w (i, j) = p (j) + q(j) + w(i, j – 1) and root r (i, j) is the value of a which minimizes c (i, j).
- Then compute c(i, j) for j – i = 2. Also compute w (i, j) and r (i, j) as in the previous step.
- Continue the process upto j – i = n. w_{on}, c_{on} and r_{on} denote weight, cost and root of optimal BST.
- Finally we can construct an OBST having root r_{on} = a, which means that key k_a is root

In general let r_{ij} is any node in an OBST, whose value is a, then its left node is $r_{i,a-1}$ and its right node is $r_{a,j}$. It is shown below in Fig. 3.7.

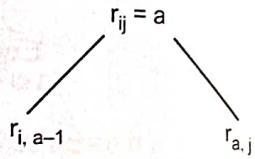


Fig. 3.7

Using this, we can construct a tree until we get $r_{ii} = 0$ at all the nodes and these are external nodes of a tree.

Let us solve few examples :

Example 3.3 : *Find an OBST using dynamic programming for n = 4 and keys (k_1, k_2, k_3, k_4) = (do, if, int, while). Given that p(1 : 4) = (3, 3, 1, 1) and q(0 : 4) = (2, 3, 1, 1, 1).*

Solution :

Step 1 : Initially, c(i, i) = 0, r (i, i) = 0 and w(i, i) = q(i) for $0 \le i \le 4$.

Hence w(0, 0) = 2, w(1, 1) = 3, w (2, 2) = w(3, 3) = w(4, 4) = 1

Step 2 :

$$w(i, j) = p(j) + q(j) + w(i, j-1).$$

$$c(i, j) = \min_{i \le a \le j} [c(i, a-1) + c(a, j)] + w(i, j)$$

$$r(i, j) = \text{value of a which minimizes c(i, j)}$$

Let us compute c(i, j) for j – i = 1.

$$w(0, 1) = p(1) + q(1) + w(0, 0) = 3 + 3 + 2 = 8$$
$$c(0, 1) = w(0, 1) + \min[c(0, 0) + c(1, 1)] = 8 + [0 + 0] = 8$$
$$r(0, 1) = 1$$
$$w(1, 2) = p(2) + q(2) + w(1, 1) = 3 + 1 + 3 = 7$$
$$c(1, 2) = w(1, 2) + \min[c(1, 1) + c(2, 2)] = 7 + [0 + 0] = 7$$
$$r(1, 2) = 2$$
$$w(2, 3) = p(3) + q(3) + w(2, 2) = 1 + 1 + 1 = 3$$
$$c(2, 3) = w(2, 3) + \min[c(2, 2) + c(3, 3)] = 3 + [0 + 0] = 3$$
$$r(2, 3) = 3$$
$$w(3, 4) = p(4) + q(4) + w(3, 3) = 1 + 1 + 1 = 3$$
$$c(3, 4) = w(3, 4) + \min[c(3, 3) + c(4, 4)] = 3 + [0 + 0] = 3$$
$$r(3, 4) = 4$$

Step 3 : Compute c(i, j) for j – i = 2.

$$w(0, 2) = p(2) + q(2) + w(0, 1) = 3 + 1 + 8 = 12$$
$$c(0, 2) = w(0, 2) + \min[c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)]$$
$$= 12 + \min [0 + 7, 8 + 0] = 12 + 7 = 19$$
$$r(0, 2) = 1$$

$w(1, 3) = p(3) + q(3) + w(1, 2) = 1 + 1 + 7 = 9$

$c(1, 3) = w(1, 3) + \min[c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)]$

$= 9 + \min [0 + 3, 7 + 0] = 9 + 3 = 12$

$r(1, 3) = 2$

$w(2, 4) = p(4) + q(4) + w(2, 3) = 1 + 1 + 3 = 5$

$c(2, 4) = w(2, 4) + \min[c(2, 2) + c(3, 4), c(2, 3) + c(4, 4)]$

$= 5 + \min [0 + 3, 3 + 0] = 5 + 3 = 8$

$r(2, 4) = 3$

Step 4 : Compute c(i, j) for j – i = 3.

$w(0, 3) = p(3) + q(3) + w(0, 2) = 1 + 1 + 12 = 14$

$c(0, 3) = w(0, 3) + \min[c(0, 0) + c(1, 3), c(0, 1) + c(2, 3), c(0, 2) + c(3, 4)]$

$= 14 + \min [0 + 12, 8 + 3, 19 + 3]$

$= 14 + \min [12, 11, 22] = 14 + 11 = 25$

$r(0, 3) = 2$

$w(1, 4) = p(4) + q(4) + w(1, 3) = 1 + 1 + 9 = 11$

$c(1, 4) = w(1, 4) + \min[c(1, 1) + c(2, 4), c(1, 2) + c(3, 4), c(1, 3) + c(4, 4)]$

$= 11 + \min [0 + 8, 7 + 3, 12 + 0]$

$= 11 + \min [8, 10, 12] = 11 + 8 = 19$

$r(1, 4) = 2$

Step 5 : Compute c(i, j) for j – i = 4.

$w(0, 4) = p(4) + q(4) + w(0, 3) = 1 + 1 + 14 = 16$

$c(0, 4) = w(0, 4) + \min[c(0, 0) + c(1, 4), c(0, 1) + c(2, 4), c(0, 2) + c(3, 4), c(0, 3) + c(4, 4)]$

$= 16 + \min [0 + 19, 8 + 8, 19 + 3, 25 + 0]$

$= 16 + \min [19, 16, 22, 25] = 16 + 16 = 32$

$r(0, 4) = 2$

All these computations can be shown using the following table :

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

- In the last step, we obtained $w_{04} = 16$, $c_{04} = 32$ and $r_{04} = 2$, which denotes that for the given keys = (do, if, int, while), an optimal BST has weight 16, cost 32, and its root is k_2 = if .
- In the above table, row i and coloum j shows the result of w(j, i + j), c(j, i + j) and r (j, i + j) respectively. Calculation proceeds row by row.

Let us construct an OBST from the above calculations.

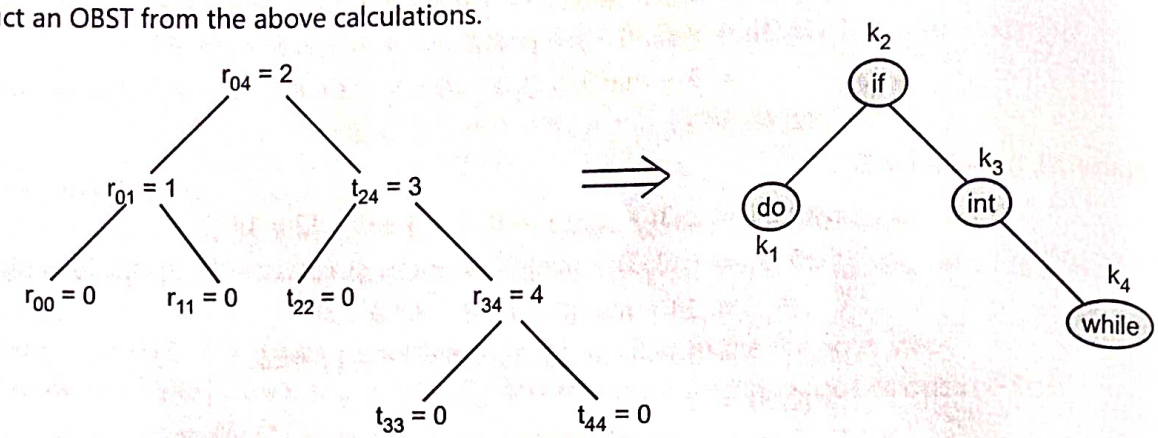


Fig. 3.8

Example 3.4 : *Find an OBST using dynamic Programming approach : N = 4, keys = (count, float, if, while). Compute w(r (i, j) and c(i, j) for $0 \le i \le j \le 4$. Given that p(1) = 1/20, p(2) = 1/5, p(3) = 1/10, p(4) = 1/20, q(0) = 1/5, q(1) = 1/ q(2) = 1/5, q(3) = 1/20 and q(4) = 1/20. Using the r(i, j)s, construct an optimal binary search tree.*

Solution :

$$p(1:4) = \left(\frac{1}{20}, \frac{1}{5}, \frac{1}{10}, \frac{1}{20}\right) = (0.05, 0.2, 0.1, 0.05)$$

$$q(0:4) = \left(\frac{1}{5}, \frac{1}{10}, \frac{1}{5}, \frac{1}{20}, \frac{1}{20}\right) = (0.2, 0.1, 0.2, 0.05, 0.05)$$

Step 1 : Initially c(i, j) = 0, r(i, j) = 0 and w(i, j) = q (i) for $0 \le i \le 4$.

Hence $w_{00} = 0.2$, $w_{11} = 0.1$, $w_{22} = 0.2$, $w_{33} = 0.05$, $w_{44} = 0.05$.

Step 2 : w(i, j) = p(j) + q(j) + w (i, j –1)

$$c(i, j) = \min_{i \le a \le j} [c(i, a-1) + c(a, j)] + w(i, j)$$

r(i, j) = value of which minimizes c(i, j)

Compute c(i, j) for j – i = 1.

w(0, 1) = p(1) + q(1) + w(0, 0) = 0.05 + 0.1 + 0.2 = 0.35

c (0, 1) = w (0, 1) + min [c(0, 0) + c(1, 1) = 0.35 + [0 + 0] = 0.35

r (0, 1) = 1

w(1, 1) = p(2) + q(2) + w(1, 1) = 0.2 + 0.2 + 0.1 = 0.5

c (1, 2) = w (1, 2) + min [c(1, 1) + c(2, 2) = 0.5 + [0 + 0] = 0.5

r (1, 2) = 2

w(2, 3) = p(3) + q(3) + w(2, 2) = 0.1 + 0.05 + 0.2 = 0.35

c (2, 3) = w (2, 3) + min [c(2, 2) + c(3, 3)] = 0.35 + [0 + 0] = 0.35

r (2, 3) = 3

w(3, 4) = p(4) + q(4) + w(3, 3) = 0.05 + 0.05 + 0.05 = 0.15

c (3, 4) w (3, 4) + min [c(3, 3) + c(4, 4) = 0.15 + [0 + 0] = 0.15

r (3, 4) = 4

Step 3 : Compute c(i , j) for j – i = 2.

w(0, 2) = p(2) + q(2) + w(0, 1) = 0.2 + 0.2 + 0.35 = 0.75

c (0, 2) = w (0, 2) + min [c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)]

= 0.75 + min[0 + 0.5, 0.35 + 0] = 0.75 + 0.35 = 1.10

r (0, 2) = 2

$w(1, 3) = p(3) + q(3) + w(1, 2) = 0.1 + 0.05 + 0.5 = 0.65$

$c(1, 3) = w(1, 3) + \min [c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)]$

$= 0.65 + \min [0 + 0.35, 0.5 + 0] = 0.65 + 0.35 = 1.00$

$r(1, 3) = 2$

$w(2, 4) = p(4) + q(4) + w(2, 3) = 0.05 + 0.05 + 0.35 = 0.45$

$c(2, 4) = w(2, 4) + \min [c(2, 2) + c(3, 4), c(2, 3) + c(4, 4)]$

$= 0.45 + \min [0 + 0.15, 0.35 + 0] = 0.45 + 0.15 = 0.60$

$r(2, 4) = 3$

Step 4 : Compute c(i, j) for j – i = 3.

$w(0, 3) = p(3) + q(3) + w(0, 2) = 0.1 + 0.05 + 0.75 = 0.90$

$c(0, 3) = w(0, 3) + \min [c(0, 0) + c(1, 3), c(0, 1) + c(2, 3), c(0, 2) + c(3, 3)]$

$= 0.9 + \min [0 + 1, 0.35 + 0.35, 1.1 + 0] = 0.9 + 0.7 = 1.6$

$r(0, 3) = 2$

$w(1, 4) = p(4) + q(4) + w(1, 3) = 0.05 + 0.05 + 0.65 = 0.75$

$c(1, 4) = w(1, 4) + \min [c(1, 1) + c(2, 4), c(1, 2) + c(3, 4), c(1, 3) + c(4, 4)]$

$= 0.75 + \min [0 + 0.6, 0.5 + 0.15, 1 + 0] = 0.75 + \min [0.6, 0.65, 1]$

$= 0.75 + 0.6 = 1.35$

$r(1, 4) = 2$

Step 5 : Compute c(i, j) for j – i = 4.

$w(0, 4) = p(4) + q(4) + w(0, 3) = 0.05 + 0.05 + 0.9 = 1.00$

$c(0, 4) = w(0, 4) + \min [c(0, 0) + c(1, 4), c(0, 1) + c(2, 4), c(0, 2) + c(3, 4), c(0, 3) + c(4, 4)]$

$= 1 + \min [0 + 1.35, 0.35 + 0.6, 1.1 + 0.15, 1.6 + 0]$

$= 1 + \min [1.35, 0.95, 1.25, 1.6]$

$= 1 + 0.95 = 1.95$

$r(0, 4) = 2$

Hence for keys (k_1, k_2, k_3, k_4) = (cout, float, if, while), an OBST has weight $w_{04} = 1$, cost $c_{04} = 1.95$ and root $r_{04} = 2$.

These calculations can be written in the table form as shown below :

	0	1	2	3	4
0	$w_{00} = 0.2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 0.1$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 0.2$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 0.05$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 0.05$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 0.35$ $c_{01} = 0.35$ $r_{01} = 1$	$w_{12} = 0.5$ $c_{12} = 0.5$ $r_{12} = 2$	$w_{23} = 0.35$ $c_{23} = 0.35$ $r_{23} = 3$	$w_{34} = 0.15$ $c_{34} = 0.15$ $r_{34} = 4$	
2	$w_{02} = 0.75$ $c_{02} = 1.1$ $r_{02} = 2$	$w_{13} = 0.65$ $c_{13} = 1$ $r_{13} = 2$	$w_{24} = 0.45$ $c_{24} = 0.6$ $r_{24} = 3$		
3	$w_{03} = 0.9$ $c_{03} = 1.6$ $r_{03} = 2$	$w_{14} = 0.75$ $c_{14} = 1.35$ $r_{14} = 2$			
4	$w_{04} = 1$ $c_{04} = 1.95$ $r_{04} = 2$				

Let us construct OBST using above calculations :

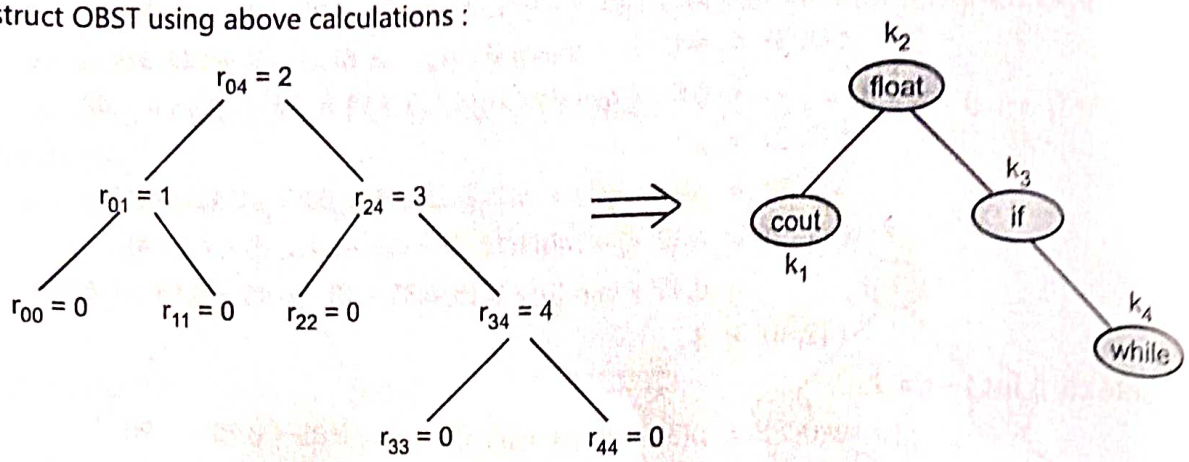


Fig. 3.9

Let us write an algorithm to construct an OBST using the above formulae :

Observe that in step 1, we get simplified formulae as

$$c(i, j) = w(i, j) + c(i, j-1) + c(j, j)$$

But in step 1,

$$j - i = 1$$

$\therefore$

$$j = i + 1$$

So we can replace j by i + 1 to get

$$c(i, i+1) = w(i, i+1) + c(i, i) + c(i+1, i+1)$$

But

$$c(i, i) = c(i+1, i+1) = 0$$

$\therefore$

$$c(i, i+1) = w(i, i+1)$$

and r(i, j) = j. So we can use one for loop which will perform initialization c(i, i), r(i, i), w(i, i) and also calculate c(i, j), w(i, j) and also calculate c(i, j), w(i, j), r(i, j) for j – i =1.

Then for j – i = 2, 3, ..., n, we can use separate loop in which we will perform all calculations. For c(i, j), we will function SearchMinValue ( ) which returns a value which minimizes value of c(i, j), and this is the value of r (i, j).

The algorithm has 3 input parameters : number of keys n, an array of probabilities p [1 : n] for successful searche array of probabilies q [0 : n] for unsuccessful searches. The output of algorithm is in the form of 3 values : weight w[cost c[0, n], root r [0, n].

Actually algorithm constructs these arrays : weight w[n : n], cost c[n : n], root r[n : n]. Assume that ends at n for [n].

Here is an algorithm :

Algorithm: DOBST (n, p, q, w, c, r)

```
begin
    for i = 0 to n - 1
    begin // Initialization
        w[i, i] = q [i]
        c [i, i] = 0;
        r [i, i] = 0;
        // for j - i = 1 (here j = i + 1)
        w[i, i + 1] = p [i + 1] + q[ i + 1] + w[i, i];
        c[i, i + 1] = w[i, i + 1]
        r[i, i + 1] = i + 1;
    end
    w[n, n] = q[n]
    c[n, n] = 0
```

```
    r[n, n] = 0
    //Computation for j – i = 2, 3, ---, n
    for k = 2 to n
        begin
        for i = 0 to n – k
        begin
            j = i + k
            w[i, j] = p[j] + q[j] + w[i, j -1]
            //search value which minimizes value of cost c(i, j).
            a = SerachMinValue (c, r, i, j)
            r[i, j] = a
            c[i, j] = w [i, j] + c[i, a – 1] + c [a, j]
        end
    end
end // end of algorithm
```

Algorithm: SearchMinValue (c, r, i, j)

```
begin
    min = ∞; // Initialize m to some larger value
    for a = i + 1 to j
    begin
        if min > (c[i, a-1] + c[a, j]) then
        begin
            min = c[i, a-1] + c [a, j]
            value = a
        end if
    end
    return value
end
```

Let us write algorithm to construct an OBST using data obtained from algorithm DOBST. Algorithm uses following data structure to store a node of a tree :

```
Struct treeNode
{
    int KeyNo, i, j;
    char KeyData;
    struct treeNode * Lchild, *Rchild;
};
```

The algorithm takes 3 input parameters : Number of keys n, an array keys [k_1, k_2, ..., k_n], and array for root r[n, n]. Assume array uses index 0 as well as n. r[0, n] denotes root of an OBST for which we have to construct remaining tree. The output of an algorithm is pointer to root node for OBST. Here is an algorithm :

Algorithm: ConstructOBST (n, keys, r)

```
    //create root node
    struct treeNode * root = new treeNode;
    root -> i = 0
```

```
    root → j = n
    root → KeyNo = r [0, n]
    root → KeyData = keys [ r[0, n]]
    root → Lchild = root → Rchild = NULL
    struct treeNode * temp = root;
    construct (temp);
    // return pointer to root node
    return root;
end
```

Algorithm: Construct (struct treeNode * temp)

```
begin
    if (temp==NULL)
        return;
    else
    begin
        //construct left child
        struct treeNode * lnode = new treeNode;
        lnode → i = temp → i
        lnode → j = (temp → KeyNo) - 1
        lnode → KeyNo = r [lnode → i, lnode → j]
        if (lnode → KeyNo != 0) then
            lnode → KeyData = keys [lnode → KeyNo]
        lnode → Lchild = lnode → Rchild = NULL;
        temp → Lchild = lnode;
        if (lnode → KeyNo==0)
            temp → Lchild = NULL
        // construct right child
        struct treeNode * rnode = new treeNode;
        rnode → i = temp → KeyNo
        rnode → j = temp → j
        rnode → KeyNo = r [rnode → i, rnode → j]
        if (rnode → KeyNo != 0) then
            rnode → KeyData = keys [rnode → KeyNo]
        rnode → Lchild = rnode → Rchild = NULL;
        temp → Lchild = rnode;
        if (rnode → KeyNo==0)
            temp → Rchild = NULL

        construct (temp → Lchild )
        construct (temp → Rchild)
    end if
end
```

The time-complexity of algorithm DOBST is $O(n^2)$ and time complexity of algorithm ConstructOBST is $O(n)$.

3.11 KNAPSACK PROBLEM

We are given n objects and a knapsack. Object i has a weight w_i and the knapsack has a capacity M. If $x_i = 1$, the object i is placed into the knapsack and a profit $p_i \cdot x_i$ is earned. If $x_i = 0$, the object is not added into the knapsack and obviously no profit is earned. The objective is to obtain a filling of the knapsack that minimizes the total profit earned. Since the capacity is M, we require the total weight of all chosen objects to be almost M. Formally, this can be stated as :

maximize $\sum_{1 \le i \le n} p_i x_i$

subject to $\sum_{1 \le i \le n} w_i x_i \le M$

and $x_i = 0/1, \quad 1 \le i \le n$

A feasible solution is any set $(x_1, x_2, \ldots, x_n)$ satisfying above equations and an optimal solution is a feasible solution for which $\sum p_i x_i$ is maximum.

Let $y_1, y_2, \ldots, y_n$ be an optimal sequence of 0/1 values for $x_1, x_2, \ldots, x_n$ respectively. If $y_1 = 0$, then $y_2, y_3, \ldots, y_n$ must constitute an optimal sequence for problem KNAP (2, n, M). If it does not then $y_1, y_2, \ldots, y_n$ is not an optimal sequence for KNAP (1, n, M).

If $y_1 = 1$, then $y_2, y_3, \ldots, y_n$ must be an optimal sequence for the problem KNAP (2, n, $M - w_1$). If it is not, then there is another 0/1 sequence $z_2, z_3, \ldots, z_n$ such that

$$\sum_{2 \le i \le n} w_i z_i \le M - w_1 \text{ and}$$

$$\sum_{2 \le i \le n} P_i z_i > \sum_{2 \le i \le n} P_i y_i$$

Hence, sequence $y_1, z_2, z_3, \ldots, z_n$ is a sequence for $\sum w_i x_i$ with a greater value.

In dynamic programming formulating the optimal sequence of knapsack problem can be achieved either in forward approach or in backward approach. Let $x_1, x_2, \ldots, x_n$ be the variables for which sequence of decisions has to be made. In the forward approach, the formulation of decision x_i is made in terms of optimal decision sequences for $x_{i+1}, \ldots, x_n$. In the backward approach, the formulation for decision x_i is in terms of optimal decision sequences for $x_1, \ldots, x_{i-1}$. In forward approach, we look "ahead" on the decision sequence $x_1, x_2, \ldots, x_n$ and in backward formulation we look "backwards" on the decision sequence $x_1, x_2, \ldots, x_n$.

For integer y such that $0 \le y \le M$, $f_i(y)$ is an ascending function.

$$0 = y_1 < y_2 < \ldots < y_k \text{ such that}$$

$$f_i(y_1) < f_i(y_2) < \ldots < f_i(y_k)$$

$$f_i(y) = -\infty \text{ for } y < y_1$$

$$f_i(y) = f(y_k) \text{ for } y \ge y_k.$$

We use the ordered set

$$f_i(y) = S^i = \{ (p, w) \mid 1 \le j \le k \} \text{ where}$$

$$p = f_i(y_j) \text{ and } w = y_j.$$

The steps to solve all knapsack problem using dynamic programming forward approach are given below :

Initially $S^0 = \{ (0, 0) \}$

$S_1^i = \{ (p, w) \mid (p - p_i), (w - w_i) \in S^i \}$

That is, to obtain S^{i+1}, we either include x_{i+1} or do not include x_{i+1}

If $x_{i+1} = 1$ (not included), then $S_1^i = S^i$.

If $x_{i+1} = 1$ (included), then the resulting states in S_1^i are obtained by adding (p_{i+1}, w_{i+1}) to each state in S^i.

3. S^{i+1} can be computed by merging and purging the states in S^i and S^i_1 together, using the dominance rule on next page:

- If S^{i+1} contains two pairs (p_a, w_a) and (p_b, w_b) where $p_a \leq p_b$ and $w_a \geq w_b$, then the (p_a, w_a) is dominated by (p_b, w_b) pair. Hence pair (p_a, w_a) is discarded. In this way, dominated tuples get purged. We can also purge all pairs (p, w) with w > M, because the knapsack capacity is M.

4. Repeat steps 2 and 3 until S^n is obtained.
5. $f_n(M) = S^n$. Using this we can find solution for KNAP (1, n, m).
6. If the last pair in S^n is (p, w) then

   set $x_n = 0$ if $(p, w) \in S^{n-1}$

   set $x_n = 1$ if $(p, w) \notin S^{n-1}$, and compute $p = p - x_n$ and $w = w - w_n$.

7. Repeat step 7 for x = n, ..., 1.

Let us solve few examples :

Example 3.5 : *Generate the sets S^i, $0 \leq i \leq 3$, for the following knapsack instance : n = 3, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$ and M = 6. Also find an optimal solution.*

Solution :

$$S^0 = \{(0, 0)\}$$

S^0_1 is obtained by adding $(p_1, w_1) = (1, 2)$ to each pair of S^0.

$$S^0_1 = \{(1, 2)\}$$

S^1 is obtained by merging and purging S^0 and S^0_1.

$$S^1 = \{(0, 0), (1, 2)\}$$

S^1_1 is obtained by adding $(p_2, w_2) = (2, 3)$ to each pair of S^1.

$$S^1_1 = \{(2, 3), (3, 5)\}$$

S^2 is obtained by merging and purging S^1 and S^1_1.

$$S^2 = \{(0, 0), (1, 2), \{(2, 3), (3, 5)\}$$

S^2_1 is obtained by adding $(p_3, w_3) = (5, 4)$ to each pair of S^2.

$$S^2_1 = \{(5, 4), (6, 6), \{(7, 7), (8, 9)\}$$

S^3 is obtained by merging and purging S^2 and S^2_1.

$$S^3 = \{(0, 0), (1, 2), \{(2, 3), (5, 4), (6, 6)\}$$

Pair (5, 4) get purged here by dominance rule. Also pairs (7, 7) and (8, 9) get purged because w > M.

Last pair in S^3 is (p, w) = (6, 6) $\notin S^2$, hence $x_3 = 1$. But $(p_3, w_3) = (5, 4)$

Hence $(p, w) = (6 - 5, 6 - 4) = (1, 2)$.

Because $(1, 2) \in S^2$ and $(1, 2) \in S^1$, set $x_2 = 0$

Because $(1, 2) \notin S^0$; set $x_1 = 1$. Hence an optimal solution for the given knapsack problem is $(x_1, x_2, x_3) = (1, 0, 1)$.

Example 3.6 : *Generate the sets S^i, $0 \leq i \leq 4$, for the following knapsack instance : n = 4, $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$, $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$.*

Solution :

$$S^0 = \{(0, 0)\}$$

By adding $(p_1, w_1) = (2, 10)$ to each pair of S^0. we get

$$S^0_1 = \{(2, 10)\}$$

By merging and purging S^0 and S_1^0. we get

$$S^1 = \{(0, 0), (2, 10)\}$$

By adding $(p_2, w_2) = (5, 15)$ to each pair of S^1.we get

$$S_1^1 = \{(5, 15), (7, 25)\}$$

By merging and purging S^1 and S_1^1. we get

$$S^2 = \{(0, 0), (2, 10), \{(5, 15), (7, 25)\}$$

By adding $(p_3, w_3) = (8, 6)$ to each pair of S^2. we get

$$S_1^2 = \{(8, 6), (10, 16), \{(13, 21), (15, 31)\}$$

By merging and purging S^2 and S_1^2, we get

$$S^3 = \{(0, 0), (8, 6), \{(10, 16), (13, 21), (15, 31)\}$$

pairs (2, 10), (5, 15), (7, 25) get purged here by the dominance rule.

By adding $(p_4, w_4) = (1, 9)$ to each pair of S^3, we get

$$S_1^3 = \{(1, 9), (9, 15), (11, 25), (14, 30), (16, 40)\}$$

By merging and purging S^3 and S_1^3, we get

$$S^4 = \{(0, 0), (8, 6), (9, 15), (10, 16), (13, 21), (14, 30), (15, 31), (16, 40)\}$$

Here pairs (1, 9) and (11, 25) get purged by the dominance rule.

mple 3.7 : *Generate the sets S^i and find an optimal solution for the following knapsack instance : n = 6, $(p_1, p_2, p_3, p_4, p_5, p_6)$ $_1, w_2, w_3, w_4, w_5, w_6) = (100, 50, 20, 10, 7, 3)$ and M = 165.*

ution :

Here $p_i = w_i$ for all i, hence each pair (p, w) = p.

$$S^0 = \{0\}$$

$$S_1^0 = \{100\}$$

$$S^1 = \{0, 100\}$$

$$S_1^1 = \{50, 150\}$$

$$S^2 = \{0, 50, 100, 150\}$$

$$S_1^2 = \{20, 70, 120, 170\}$$

$$S^3 = \{0, 20, 50, 70, 100, 120, 150\}$$

Here 170 is purged because 170 > M.

$$S_1^3 = \{10, 30, 60, 80, 110, 130, 160\}$$

$$S^4 = \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\}$$

$$S_1^4 = \{7, 17, 27, 37, 57, 67, 77, 87, 107, 117, 127, 137, 157, 167\}$$

$$S^5 = \{0, , 7, 10, 17, 20, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, 107, 110, 117, 120, 127, 130, 137, 150, 157, 160\}$$

$$S_1^5 = \{3, 10, 13, 20, 23, 30, 33, 40, 53, 60, 63, 70, 73, 80, 83, 90, 103, 110, 113, 120, 123, 130, 133, 140, 153, 160, 163\}$$

$$S^6 = \{0, 3, 7, 10, 13, 17, 20, 23, 27, 33, 37, 40, 50, 53, 57, 60, 63, 67, 70, 73, 77, 80, 83, 87, 90, 100, 103, 107, 110, 113, 117, 120, 123, 127, 130, 133, 137, 140, 150, 153, 157, 160, 163\}$$

The value of $F_6(165)$ can be determined form S^6. Last tuple in S^6 is $p = w = 163 \notin S^5$ Hence $x_6 = 1$. But $p_6 = w_6 = 3$ hence $p - p_6 = 163 - 3 = 160 \in S^5$ and also $160 \in S^4$. Hence $x_5 = 0$. Now $160 \notin S^3$, hence $x_4 = 1$. But $p_4 = 10$, hence $p - p_4 = 160 - 10 = 150 \in S^3$ and also $150 \in S^2$, hence $x_3 = 0$.

But $150 \notin S^1$, hence $x_2 = 1$.

But $p_2 = 50$. Hence $p - p_2 = 150 - 50 = 100 \in S^1$ and $100 \notin S^0$, hence $x_1 = 1$.

Hence an optimal solution is

$$(x_1, x_2, x_3, x_4, x_5, x_6) = (1, 1, 0, 1, 0, 1).$$

Let us write an algorithm DKnapsack which take 4 input parameters : an array p[1 : n] for profits, an array [1 : n] for weights, number of an objects n and maximum capacity of Knapsak M.

Algorithm: Dknapsack (p, w, n, M)

```
begin
    S^0 = {(0, 0)}
    for i = 0 to n - 1
    begin
        S_1^i = {(p, w) | for all (x, y) ∈ S^i, compute (p, w) = (x + p_{i+1}, y+w_{i+1})}
        S^{i+1} = MergeAndPurge (S^i, S_1^i)
    end
-   Let (p_x, w_x) is last pair in S^n.
-   (p_y, w_y) = (p' + p_n, w' + w_n) where w' is the largest w in any pair in s^n such that w + w_n ≤ M.
    // Trace back for x_n, x_{n-1}, ...., x_1
    if (p_x > p_y) then
        x_n = 0
    else
        x_n = 1
    TraceBack for (x_{n-1}, ...., x_1)
end
```

The complexity of the algorithm depends on how S^i and S_1^i are represented.

3.12 ALL PAIRS SHORTEST PATH (FLOYD-WARSHALL ALGORITHM)

- In this section, we are going to study an algorithm known as Floyd-Warshall algorithm. It uses dynamic programming approach. Let G(V, E) is a directed graph. Assume that negative-weight edges are allowed in the graph. The algorithm considers the intermediate vertices of a shortest path. An intermediate vertex of a simple path $P = \{v_1, v_2, ..., v_m\}$ is any vertex of P other than v_1 or v_m. The intermediate vertex $\in \{v_2, v_3, ..., v_{m-1}\}$.
- For any pair of vertices $i, j \in V$, let P is a minimum-weight path among (i, j) and its intermediate vertices $\in \{1, 2, ..., k\}$.
  - ➢ If k is not an intermediate vertex of path p, then shortest path from i to j using intermediate vertices $\{1, 2, ..., k-1\}$ is same as shortest path from i to j using intermediate vertices {1, 2, ..., k}.
  - ➢ If k is an intermediate vertex of path P, then P is made up of two shortest paths : First path from i to k and second path from k to j. Both these paths are shortest paths which use intermediate vertices from $\{1, 2, ..., k-1\}$.

To compute all pairs shortest paths, use the following steps :

- Represent the graph using matrix W which follows the following recurrence :

$$w_{ij} = \begin{cases} 0 & , \text{if } i = j \\ \text{weight}(i, j) & , \text{if } i \neq j \text{ and edge } \langle i, j \rangle \in E \\ \infty & , \text{if } i \neq j \text{ and edge } \langle i, j \rangle \notin E \end{cases}$$

- Let $D(n) = d_{ij}(n)$, where $d_{ij}(k)$ denotes the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, ..., k}. When k = 0, d $d_{ij}(0)$, w_{ij}.

$$d_{ij}(k) = \begin{cases} w_{ij} & , \text{if } k = 0 \\ \min[d_{ij}(k-1), d_{ik}(k-1), d_{kj}(k-1)] & , \text{if } k \geq 1 \end{cases}$$

The algorithm All Pairs Shortest Paths takes two inputs : the matrix W[n, n] and number of vertices n. It returns matrix D(n) of shortest-path lengths as output. The algorithm is as given below :

Algorithm: All Pairs Shortest Paths (W, n)

```
begin
  D(0) = W
  for k = 1 to n
    for i = 1 to n
      for j = 1 to n
        d_ij(k) = min [d_ij(k-1), d_ik(k-1) + d_kj(k-1)]
      end for
    end for
  end for
  return D(n)
end
```

Analysis of All Pairs Shortest Paths Algorithm:

Computing time of this algorithm depends on the computing time of $d_{ij}(k)$ which is constant, that is, O(1). Hence computing time of algorithm is $O(n^3)$.

Let us solve few examples to find all pairs shortest paths.

Examples 3.8 : *Compute all pairs shortest path lengths in the following graph of Fig. 3.10.*

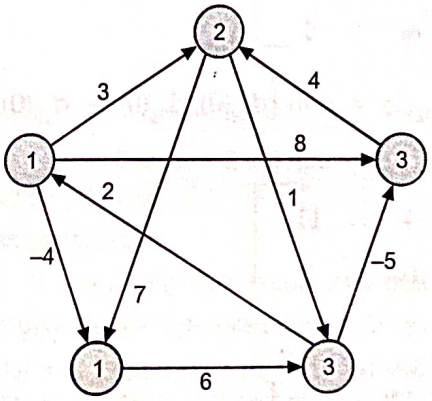


Fig. 3.10

Solution :

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

$$D(2) = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad D(3) = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

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

Example 3.9 : *Compute all pairs shortest path lengths for the following graph in Fig. 3.11.*

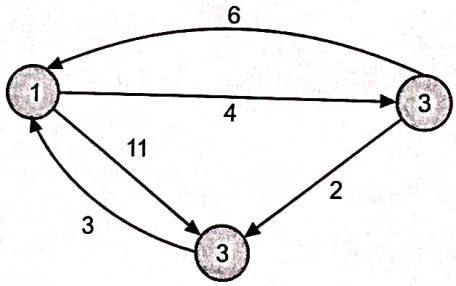


Fig. 3.11

Solution :

$$D(0) = W = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$d_{32}(1) = \min \left[d_{32}(0), d_{31}(0) + d_{12}(0) \right]$$

$$= \min [\infty, 3 + 4] = 7$$

$$D(1) = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$d_{13}(2) = \min \left[d_{13}(1), d_{12}(1) + d_{23}(1) \right]$$

$$= \min [11, 4 + 2] = 6$$

$$D(2) = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$d_{21}(3) = \min \left[d_{21}(2), d_{23}(2) + d_{31}(2) \right]$$

$$= \min [6, 2 + 3] = 5$$

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

3.13 TRAVELLING SALESPERSON PROBLEM

Let G = (V, E) be a directed graph, with each edge having a cost. A directed simple cycle which includes every vertex in graph is called a tour of graph G. The sum of the cost of all the edges in the tour is the cost of the tour.

Problem Definition : The travelling salesperson problem is to determine a tour of minimum cost.

Application of Travelling Salesperson Problem :

- Consider a graph in which a vertex represents city and an edge represents distance between two cities. If a sales person wants to visit all the cities for selling items starting from and ending at home city, then the route taken by a sales person is a tour and he is interested in finding a smallest route.
- A company wants to use a robot to pack a produced item in a box at each machine after every hour. There are many machines. If a machine denotes vertex of a graph, then time required to pack an item is same at each machine. But the time required to move from one machine to other is cost of the edge which connects those two machines. So the route taken by a robot is a tour and company is interested in finding a tour which requires minimum time.

Let us see how travelling salesperson problem can be solved.

Let G = (V, E) be a directed graph. Variable c_{ij} denotes cost of an edge $< i, j >$. If $<i, j> \in E$, then $c_{ij} > 0$. If $< i, j > \notin E$, then $c_{ij} = \infty$. Let $|v| = n$ for $n > 1$.

A path which starts and ends at the same vertex visiting all the vertices in the graph is called a tour of graph. Consider a cyclic path which starts and ends at vertex 1.

This path consists of an edge $<1, k>$ for $k \in V - \{1\}$ and a shortest path from k to 1 visiting each vertex in $V - \{1, k\}$ exactly once. Hence the principle of optimality holds. From the principle of optimality, we obtain the function $g(1, V - \{1\})$ which gives minimum length of tour, as

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

In general, the length of a shortest path which starts at i, goes through all vertices in set S and terminates at vertex 1 is given by g(i, S). Here $i \neq 1$, $i \notin S$, $1 \notin S$. Then g (i, S) is given by

$$g(i, S) = \min_{k \in S} \{c_{ij} + g(k, S - \{k\})\} \quad \ldots (2)$$

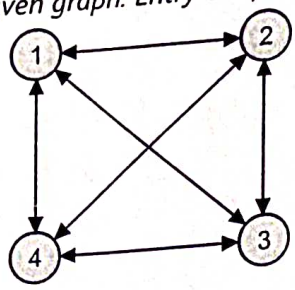
Using equation (2), we can

- First compute g (i, S) with |S| = 0 i.e.
  Compute $g(i, \phi) = c_{i1}$ for $2 \le i \le n$.
- Then compute g (i, S) with |S| = 1.
- Then compute g (i, S) with |S| = 2 and so on upto | S | = n – 2.
- Finally compute g (1, V – {1}) using equation (1).
- At each stage which computes g (i, S) remember the value of j that minimizes the cost of g (i, S). Let d (i, S) denotes this decision. These d values give us the optimal tour, i.e. the minimum-length path.

From equation (2), it can be observed that if |S| = x, then x – 1 comparisons are done to compute g (i, S).

So the time complexity of travelling salesperson problem is $\theta(n^2 2^n)$. But the space complexity is very high which is $O(n2^n)$. This is major drawback of this algorithm if it is solved using dynamic programming method.

Example 3.10 : *Consider the following Figures. Fig. 3.12 (a) shows a directed graph. Fig. 3.12 (b) shows a matrix wh denotes cost of edges in given graph. Entry 0 represents no edge.*



(a)

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

(b)

Fig. 3.12

Find an optimal tour from vertex 1 to 1 in the given graph.

Solution :

(1) First compute g (i, ϕ) for each vertex i $\neq$ 1

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

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

$$g(2, \{3\}) = c_{23} + g(3, \phi) = 9 + 6 = 15$$
$$g(2, \{4\}) = c_{24} + g(4, \phi) = 10 + 8 = 18$$
$$g(3, \{2\}) = c_{32} + g(2, \phi) = 13 + 5 = 18$$
$$g(3, \{4\}) = c_{34} + g(4, \phi) = 12 + 8 = 20$$
$$g(4, \{2\}) = c_{42} + g(2, \phi) = 8 + 5 = 13$$
$$g(4, \{3\}) = c_{43} + g(3, \phi) = 9 + 6 = 15$$

(3) Compute g (i, S) with |S| = 2 and i $\neq$ 1, 1 $\notin$ S, i $\notin$ S.

$$g(2, \{3, 4\}) = \min [c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})]$$
$$= \min [9 + 20, 10 + 15] = \min [29, 25] = 25$$

$\therefore$ $d(2, \{3, 4\}) = 4$ $\because$ vertex 4 minimizes cost of g (2, {3, 4}),

$$g(3, \{2, 4\}) = \min [c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})]$$
$$= \min [13 + 18, 12 + 13] = \min [31, 25] = 25$$

$\therefore$ $d(3, \{2, 4\}) = 4$

$$g(4, \{2, 3\}) = \min [c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})]$$
$$= \min [8 + 15, 9 + 18] = \min [23, 27] = 23$$

$\therefore$ $d(4, \{2, 3\}) = 2$

(4) Compute g(i, S) with |S| = 3, i.e. g (1, V – {1}).

$$g(1, \{2, 3, 4\}) = \min [c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}) c_{14} + g(4, \{2, 3\})]$$
$$= \min [10 + 25, 15 + 25, 20 + 23] = \min [35, 40, 43] = 35$$

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

Hence an optimal tour has cost 35. Let us determine an optimal path.

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

Hence an optimal tour is 1 – 2 – 4 – 3 – 1.

SOLVED EXERCISE

I. Long questions answers :

1. Let n = 4, (a_1, a_2, a_3, a_4) = (do, if, read, while)

Let $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ and $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$.

Construct OBST.

Ans. : Initially we have $w_{ij} = q_i$ and $c_{ii} = 0$ and $r_{ii} = 0$, $0 \le i \le 4$.

Using equations

$$c_{ij} = w_{ij} + c_{i, k-1} + c_{k, j} \text{ and} \quad \ldots (1)$$

$$c_{i, k-1} + c_{k, j} = \min_{i < l \le j} \{c_{i, l-1} + c_{i, j}\} \quad \ldots (2)$$

(a)
$$w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8$$
$$c_{01} = w_{01} + \min \{c_{00} + c_{11}\} = 8$$
$$r_{01} = 1$$

(b)
$$w_{12} = p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7$$
$$c_{12} = w_{12} + \min \{c_{11} + c_{22}\} = 7$$
$$r_{12} = 2$$

(c)
$$w_{23} = p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3$$
$$c_{23} = w_{23} + \min \{c_{22} + c_{33}\} = 3$$
$$r_{34} = 4$$

(d)
$$w_{34} = p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3$$
$$c_{34} = w_{34} + \min \{c_{33} + c_{44}\} = 3$$
$$r_{34} = 4$$

Knowing $w_{i, i+4}$ and $c_{i, i+4}$, $0 \le i < 4$ we can again use equation (1) and (2) to compute $w_{i, i+2}$, $c_{i, i+2}$, $r_{i, i+2}$, $0 \le i \le 3$. This process may be repeated until w_{04}, c_{04} and r_{04} are obtained. From above table we see that $c_{04} = 3$ is the minimal cost of BST for a_1 to a_4. The root of tree T_{04} is a_2. Hence, the left subtree is T_{01} and the right subtree T_{24}. T_{01} has root a_{01} and subtrees T_{00} and T_{11}. T_{24} has root a_3; its left subtree is therefore T_{22} and right subtree T_{34}. Thus, with the data is table, we get T_{04} as shown in figure.

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

The OBST :

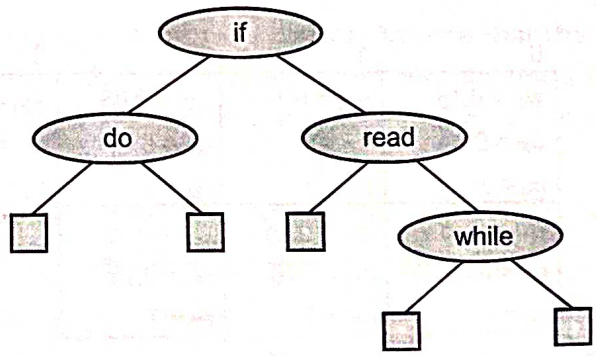


Fig. 3.13

2. Construct OBST for

(a_1, a_2, a_3, a_4) = (end, goto, print, read)

(p_1, p_2, p_3, p_4) = (1/20, 1/5, 1/10, 1/20)

$(q_0, q_1, q_2, q_3, q_4)$ = (1/5, 1/10, 1/5, 1/20, 1/20)

Let us multiply all probabilities by 20 and yield following for convenience :

(p_1, p_2, p_3, p_4) = (1, 4, 2, 1)

$(q_0, q_1, q_2, q_3, q_4)$ = (4, 2, 4, 1, 1)

The table constructed :

	0	1	2	3	4
0	$w_{00} = 4$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 2$ $c_{11} = 6$ $r_{11} = 0$	$w_{22} = 4$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 7$ $c_{01} = 7$ $r_{01} = 1$	$w_{12} = 10$ $c_{12} = 10$ $r_{12} = 2$	$w_{23} = 7$ $c_{23} = 7$ $r_{23} = 7$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 15$ $c_{02} = 22$ $r_{02} = 2$	$w_{13} = 13$ $c_{13} = 20$ $r_{13} = 2$	$w_{24} = 9$ $c_{24} = 12$ $r_{24} = 3$		
3	$w_{03} = 18$ $c_{03} = 32$ $r_{03} = 2$	$w_{14} = 15$ $c_{14} = 27$ $r_{14} = 2$			
4	$w_{04} = 20$ $c_{04} = 39$ $r_{04} = 2$				

Hence OBST :

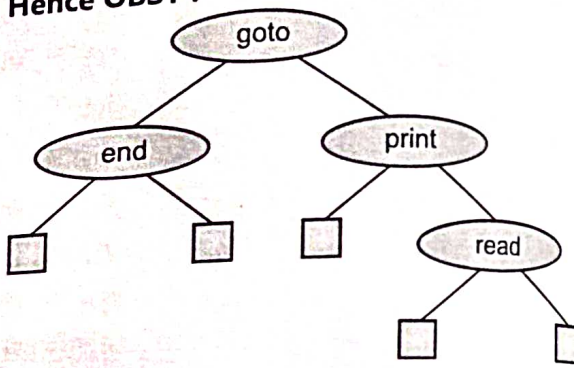


Fig. 3.14

3. Let n = 3 and $\{a_1, a_2, a_3\}$ = {do, if, while}

Let p (1 : 3) = (0.5, 0.1, 0.05) and

q (0 : 3) = (0.15, 0.1, 0.05, 0.05)

Compute and construct OBST for above values.

Ans. : Initially we have $W_{(i, i)} = q_{(i)}$, $c_{(i, i)}$ and $r_{(i, i)} = 0$ to 3

Using $\min \{c_{(i, k-1)} + c_{(k,j)}\}$ for k = i to j + $w_{(i,j)}$

and $w(i, j) = p(j) + q(j) + w(i, j-1)$

	0	1	2	3
0	$w_{00} = 0.15$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 0.1$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 0.05$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 0.05$ $c_{33} = 0$ $r_{33} = 0$
1	$w_{01} = 0.75$ $c_{01} = 0.75$ $r_{01} = 1$	$w_{12} = 0.25$ $c_{12} = 0.25$ $r_{12} = 2$	$w_{23} = 0.15$ $c_{23} = 0.15$ $r_{23} = 3$	
2	$w_{02} = 0.9$ $c_{02} = 1.15$ $r_{02} = 1$	$w_{13} = 0.35$ $c_{13} = 0.5$ $r_{13} = 2$		
3	$w_{03} = 1$ $c_{03} = 1.5$ $r_{03} = 1$			

The OBST :

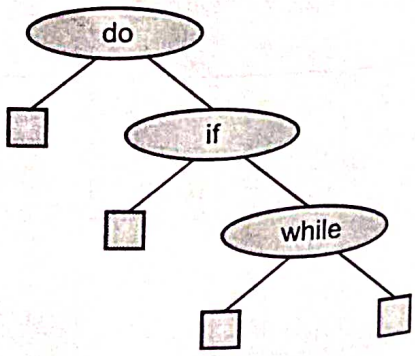


Fig. 3.15

EXERCISE

1. Explain dynamic programming approach.
2. State the principle of optimality and explain it in brief.
3. Construct OBST for (a_1, a_2, a_3) = (int, char, float). Let $p(1:3) = (0.5, 0.1, 0.05)$ and $q(0:3) = (0.15, 0.1, 0.05, 0.05)$. Also compute w(i, j), r(i, j,) and c(i, j).
4. Generate sets S^i, $0 \le i \le 4$ when weights $(w_1, w_2, w_3, w_4) = (1, 11, 21, 23)$ and profits $(p_1, p_2, p_3, p_4) = (11, 21, 31, 33)$.
5. Write control abstraction for following algorithm strategies.
   (a) Divide and Conquer
   (b) Greedy Method
   (c) Dynamic programming
6. Compare Greedy and Dynamic strategies.

UNIVERSITY QUESTION BANK

Apply dynamic programming to the problem optimal binary search tree. Assume that the given set of identifiers is $\{a_1, a_2, ..., a_n\}$ with $a1 < a_2 < ... < a_n$. Let P (i) be the probabilities with which we shall be searching for ai such that $1 \le i \le n$. Let q(i) be the probability that the identifier X being searched for, is such that $a(i) < X < a(i+1)$ for $0 \le i \le n$. Obtain an algorithm which computes the cost of optimal binary search free. Show details of each step of design.

Let n = 3 and (a_1, a_2, a_3) = (do, if, while)

Let p (1 : 3) = (05, 01, 0.05) and Let (0 : 3) = (0.15, 0.1, 0.05, 0.05)

Compute OBST for above values.

(a) Consider 0/1 knapsack problem. N = 3, W = (4, 6, 8), P = (10, 12, 15). Using dynamic programming device the algorithm for the same.

(a) Apply dynamic programming to the problem of obtaining optimal binary Search tree. Assume that the given set of I identifiers is $\{a_1, a_2, ..., a_n\}$ with $a_1 < a_2 ... < a_n$. Let p(i) be the probabilities with which a1 is searched such that $1 \le i \le n$. Let q (i) be the probability with which identifiers x is searched for, where $a(i) < x < a(i+1)$ for $0 \le i \le n$. Obtain algorithm to compute cost of optimal binary search tree.

(b) Let n = 3 and {a1, a2, a3} = {do, if, while}

p (1:3) = (0.5, 0.1, 0.05) and

q (0:3) = (0.15, 0.1, 0.05, 0.05)

Compute and construct OBST for above values.

Apply dynamic programming to the problem of obtaining an optimal binary search tree. Assume that the given set of identifiers is $\{a_1, a_2, ..., a_n\}$ with $a_1 < a_2 ... < a_n$. Let P(i) be the probability with which we shall be searching for as such that $i \le n$. Let Q(i) be the probability that the identifier X being searched for is such that $a_1 < x < a_i + 1$ with $0 \le i \le n$.

Develop an algorithm which computer the cost C(i, j) of binary search tree T_{ij} for identifiers a1+1, ...a_j and also computers R(i, j) the root T_{ij}. Assign w (i, j) to be the weight of T_{ij}. Where $W(i, j) = Q(i) + \sum_{L=I+1} Q(i) + P(L)$

6. (a) Write general template for dynamic algorithms.
   (b) Devise a ternary search algorithm which first tests the element at position $n_{/3}$ for equality with some value x which is to be searched and then possibly checks at 2n/3 either discovering x or reducing the set size to one third of the original. Compute time complexity of algorithm. Compare this with binary search algorithm.
7. (a) Comment on : "The solution to the problem by dynamic programming relies on a useful principle called the principle of optimality"

(b) Consider 0/1 Knapsack problem. Using dynamic programming approach, devise the recurrence relation for th problem and solve the same for following values :

N = 3, W = (4, 6, 8), P = (10, 12, 15)

Determine the optimal profit for the Knapsack capacity 7, 8, 10, 14.

8. (a) Using OBST compute W (i, j) R (i, j), $0 \le i, j \le 4$ for the identifier set (a_1, a_2, a_3, a_4) = (end, goto, print, stop) with (1:4) = (1/20, 1/5, 1/10, 1/20) and Q (0 : 4) = (1/5, 1/10, 1/5, 1/20,1/20). Using the (i, j) s, construct the optir binary search tree.

(b) Show that the computing time of algorithm OBST is O (n^2).

9. (a) What does dynamic programming have in common with divide-and-conquer ? What is a principal differe between the two techniques?

(b) Write dynamic algorithm for "The multistage graph problem' in which a minimum cost path from sourc destination to too be computed for multistage graph.

10. Write algorithm for OBST (Optimal Binary Search Tree).

11. Given an n-by-m chocolate bar, you need to break it into nm 1-by-1 pieces.

You can break bar only in straight line, and you can break only one bar at time. Design an algorithm that solve problem with the smallest number of bar breaks.

12. Write dynamic based algorithm for computing minimum cost from source s to destination t in multistage graph

13. (a) What does dynamic programming have in common with divide-and-conquer ?

(b) What is a principle difference between the two ?

(c) What is principle of optimality ?

(d) Design a dynamic programming algorithm for the : change-making problem : given an amount n and unli quantities of coins of each of the denominations $d_1, d_2, ..., d_m$, find the smallest number of coins that add n or indicate that the problem does a solution. Will your algorithm work correctly even when the numb coins of a participles denomination is limited ?

14. Apply dynamic programming to the problem of obtaining an optimal binary search tree. Assume that the give of identifiers is $\{a_1, a_2, ... a_n\}$ with $a_1 < a_2 < ..... < a_n$. Let P(i) be the probabilities with which we shall be searching such that 1 <= i <= n. Let Q(i) be the probability that the identifier X being searched for i's such that $a_i < X$ with 0 <= i <= n. Develop an algorithm which computes the cost C(i+j) of optimal binary search tree identifiers $a_{i+1}, ...a_j$ and also computes R(i, j) the root of T_{ij}. Assign W(i, j) to be the weight of T_{ij}.

15. (a) Give an algorithm for solving 0/1 knapsack using Dynamic Programming. Compute Time complexity

(b) Compare binary search algorithm with a ternary search algorithm.

✠ ✠ ✠