UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

3.1 COMPUTING A BINOMIAL COEFFICIENT

Dynamic Programming Binomial Coefficients

Dynamic Programming was invented by Richard Bellman, 1950. It is a very general technique for solving optimization problems.

Dynamic Programming requires:

1. Problem divided into overlapping sub-problems
2. Sub-problem can be represented by a table
3. Principle of optimality, recursive relation between smaller and larger problems

Compared to a brute force recursive algorithm that could run exponential, the dynamic programming algorithm runs typically in quadratic time. The recursive algorithm ran in exponential time while the iterative algorithm ran in linear time.

Computing a Binomial Coefficient

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming.

Binomial coefficients are represented by $C(n, k)$ =n! / (k! (n-k)!) or $\binom{n}{k}$ and can be used to represent the coefficients of a binomial:

$(a + b)^n = C(n, 0)a^n b^0 + \ldots + C(n, k)a^{n-k}b^k + \ldots + C(n, n)\, a^0 b^n$

The recursive relation is defined by the prior power

$C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k)$ for $n > k > 0$ with initial Condition $C(n, 0) = C(n, n) = 1$

Dynamic algorithm constructs a *nxk* table, with the first column and diagonal filled out using the Initial Condition. Construct the table:

	k						
		0	**1**	**2**	**...**	***k*-1**	***k***
	0	1					
	1	1	1				
	2	1	2	1			
n	**. . .**						
	k	1					1
	. . .						
	***n*-1**	1				$C(n\text{-}1, k\text{-}1)$	$C(n\text{-}1, k)$
	n	1					$C(n, k)$

The table is then filled out iteratively, row by row using the recursive relation.

Algorithm *Binomial*(n, k)

```
for i ← 0 to n do  // fill out the table row wise
    for i = 0 to min(i, k) do
        if j==0 or j==i then C[i, j] ← 1  // Initial Condition
        else C[i, j] ← C[i-1, j-1] + C[i-1, j]  // recursive relation
return C[n, k]
```

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled out for $i < k$ and remaining part of the table is filled out across the entire row.

$A(n, k)$ = sum for upper triangle + sum for the lower rectangle

$= \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=1}^{n} \sum_{j=1}^{k} 1$

$= \sum_{i=1}^{k} (i-1) + \sum_{i=1}^{n} k$

$= (k-1)k/2 + k(n-k) \in \Theta(nk)$

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Example: Relation of binomial coefficients and pascal's triangle.

A formula for computing binomial coefficients is this:

$$\binom{n}{m} = \frac{n!}{(n-m)!m!}$$

Using an identity called Pascal's Formula a recursive formulation for it looks like this:

$$\binom{n}{m} = \begin{cases} 1 & \text{if } m = 0 \\ 1 & \text{if } n = m \\ \binom{n-1}{m} + \binom{n-1}{m-1} & \text{otherwise} \end{cases}$$

This construction forms Each number in the triangle is the sum of the two numbers directly above it.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Finding a binomial coefficient is as simple as a lookup in Pascal's Triangle.

Example: $(x+y)^7 = 1 \cdot x^7y^0 + 7 \cdot x^6y^1 + 21 \cdot x^5y^2 + 35 \cdot x^4y^3 + 35 \cdot x^3y^4 + 21 \cdot x^2y^5 + 7 \cdot x^1y^6 + 1 \cdot x^0y^7$

$= x^7 + 7x^6y + 21x^5y^2 + 35x^4y^3 + 35x^3y^4 + 21x^2y^5 + 7xy^6 + y^7$

3.2 WARSHALL'S AND FLOYD' ALGORITHM

Warshall's and Floyd's Algorithms: Warshall's algorithm for computing the transitive closure (there is a path between any two nodes) of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on dynamic programming.

WARSHALL'S ALGORITHM (All-Pairs Path Existence Problem)

A **directed graph** (or digraph) is a graph, or set of vertices connected by edges, where the edges have a direction associated with them.

An **Adjacency matrix** A = $\{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex.

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

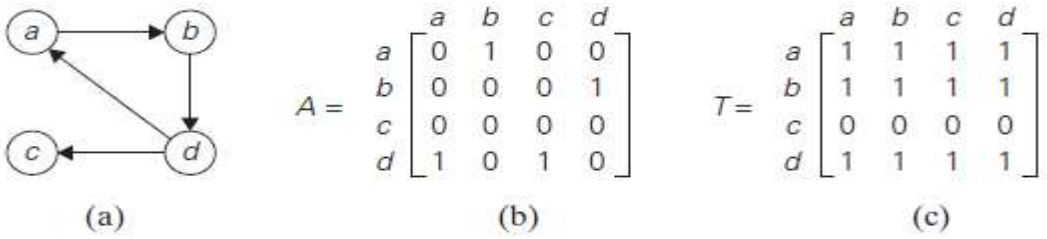


FIGURE 3.1 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

The transitive closure of a digraph can be generated with the help of depth-first search or breadth-first search. Every vertex as a starting point yields the transitive closure for all.

Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices: $R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}$.

The element $r_{ij}^{(k)}$ in the *i*th row and *j*th column of matrix $R^{(k)}$ *(i, j* = 1, 2, . . . , *n, k* = 0, 1, . . . , *n)* is equal to 1 if and only if there exists a directed path of a positive length from the *i*th vertex to the *j*th vertex with each intermediate vertex, if any, numbered not higher than *k.*

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

- The series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph.
- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate. it may contain more 1's than $R^{(0)}$.
- The last matrix in the series, R(n), reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.
- In general, each subsequent matrix in series has one more vertex to use as intermediate for its paths than its predecessor.
- The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

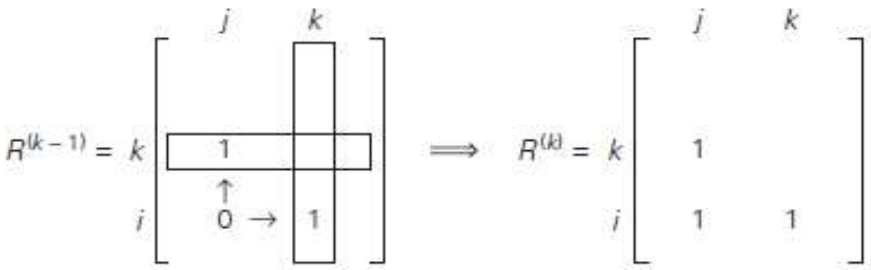


FIGURE 3.2 Rule for changing zeros in Warshall's algorithm.

All the elements of each matrix $R^{(k)}$ is computed from its immediate predecessor $R^{(k-1)}$. Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k.

The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{ik}^{(k-1)} = 1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k-1$ (hence, $r_{kj}^{(k-1)} = 1$).

Thus the following formula generas the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

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

ApplyingWarshall's algorithm by hand:

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

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

//ImplementsWarshall's algorithm for computing the transitive closure

//Input: The adjacency matrix A of a digraph with n vertices

//Output: The transitive closure of the digraph

$R^{(0)} \leftarrow A$

for k←1 **to** n **do**

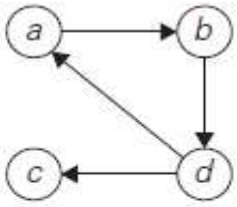
  for i ←1 **to** n **do**

    for j ←1 **to** n **do**

      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or ($R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j]$)

return $R^{(n)}$

Warshall's algorithm's time efficiency is only $\Theta(n^3)$. Space efficiency is $\Theta(n^2)$. i.e matrix size.



$R^{(0)}$ =

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

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

$R^{(1)}$ =

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

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

$R^{(2)}$ =

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

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

$R^{(3)}$ =

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

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

$R^{(4)}$ =

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

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

FIGURE 3.3 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

FLOYD'S ALGORITHM (All-Pairs Shortest-Paths Problem)

Floyd'sl algorithm is an algorithm for finding shortest paths for all pairs in a weighted connected graph (undirected or directed) with (+/-) edge weights.

A **distance matrix** is a matrix (two-dimensional array) containing the distances, taken pairwise, between the vertices of graph.

The lengths of shortest paths in an n × n matrix D called the distance matrix: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex.

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

Floyd's algorithm computes the distance matrix of a weighted graph with *n* vertices through a series of $n \times n$ matrices:

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

The element $d_{ij}^{(k)}$ in the *i*th row and the *j*th column of matrix $D^{(k)}$ *(i, j* = 1, 2, . . . , *n, k* = 0, 1, . . . , *n)* is equal to the length of the shortest path among all paths from the *i*th vertex to the *j*th vertex with each intermediate vertex, if any, numbered not higher than *k.*

Steps to compute $D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}$

- The series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is simply the weight matrix of the graph.
- As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$.
- The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix.

Let $d_{ij}^{(k)}$ be the element in the ith row and the jth column of matrix $D^{(k)}$. This means that $d_{ij}^{(k)}$ is equal to the length of the shortest path among all paths from the ith vertex v_i to the jth vertex v_j with their intermediate vertices numbered not higher than k.

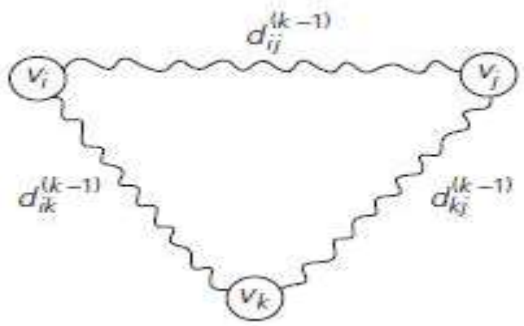


FIGURE 3.4 Underlying idea of Floyd's algorithm.

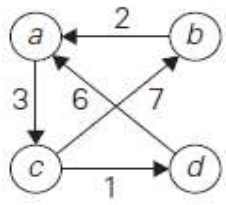
The length of the shortest path can be computed by the following recurrence:

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

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

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

Floyd's Algorithm's time efficiency is only $\Theta(n^3)$. Space efficiency is $\Theta(n^2)$. i.e matrix size.



$D^{(0)} =$

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

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

$D^{(1)} =$

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

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

$D^{(2)} =$

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

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

$D^{(3)} =$

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

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

$D^{(4)} =$

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

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

FIGURE 3.5 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

3.3 OPTIMAL BINARY SEARCH TREES

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

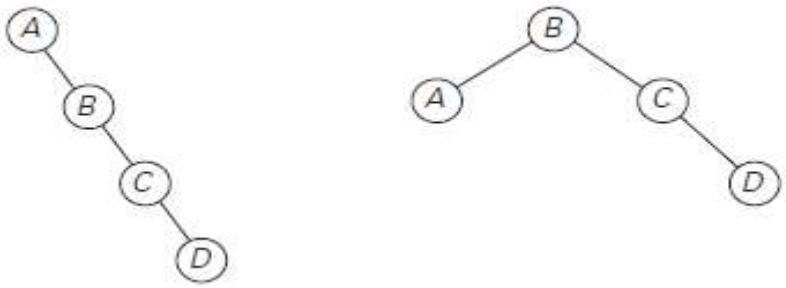


FIGURE 3.6 Two out of 14 possible binary search trees with keys *A, B, C,* and *D*.

Consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively. Figure 3.6 depicts two out of 14 possible binary search trees containing these keys.

The average number of comparisons in a successful search in the first of these trees is 0.1 . 1+ 0.2 . 2 + 0.4 . 3+ 0.3 . 4 = 2.9, and for the second one it is 0.1 . 2 + 0.2 . 1+ 0.4 . 2 + 0.3 . 3= 2.1. Neither of these two trees is optimal.

The total number of binary search trees with n keys is equal to the nth **Catalan number**,

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

c(n)=(2n)!/(n+1)!n!

Let $a_1, \ldots, a_n$ be distinct keys ordered from the smallest to the largest and let $p_1, \ldots, p_n$ be the probabilities of searching for them. Let $C(i, j)$ be the smallest average number of comparisons made in a successful search in a binary search tree T_i^j made up of keys $a_i, \ldots, a_j$, where i, j are some integer indices, $1 \le i \le j \le n$.

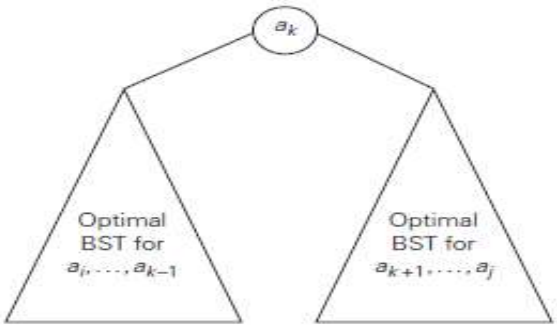


FIGURE 3.7 Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^{j}.

Consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree (Figure 3.7), the root contains key a_k, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree T_{k+1}^{j} contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged.

If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

$$C(i, j) = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\}$$

$$= \min_{i \le k \le j} \{\sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s\}$$

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

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

We assume in above formula that C(i, i − 1) = 0 for 1≤ i ≤ n + 1, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that C(i, i) = p_i for 1≤ i ≤ n, as it should be for a one-node binary search tree containing a_i.

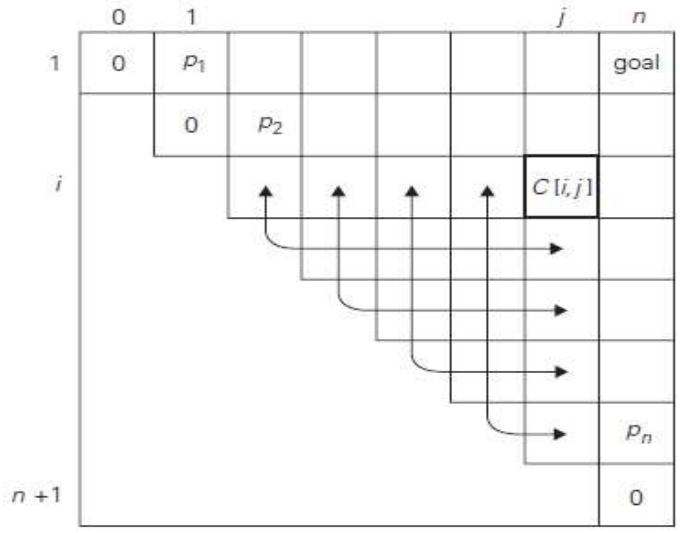


FIGURE 3.8 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

The two-dimensional table in Figure 3.8 shows the values needed for computing C(i, j). They are in row i and the columns to the left of column j and in column j and the rows below row i. The arrows point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of C(i, j). This suggests filling the table along its diagonals, starting with all zeros on the main diagonal and given probabilities p_i, $1 \le i \le n$, right above it and moving toward the upper right corner.

ALGORITHM OptimalBST(P [1..n])

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

return C[1, n], R

The algorithm's space efficiency is clearly quadratic, ie, **:** $\Theta(n^3)$; the time efficiency of this version of the algorithm is cubic. It is Possible to reduce the running time of the algorithm to $\Theta(n^2)$ by taking advantage of monotonicity of entries in the root table, i.e., R[i,j] is always in the range between R[i,j-1] and R[i+1,j]

EXAMPLE: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables are:

main table

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

root table

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

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

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

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

We arrive at the following final tables:

main table

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

root table

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

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since R(1, 4) = 3, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D. To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since R(1, 2) = 2, the root of the optimal tree containing A and B is B, with A being its left child (and the root of the one node tree: R(1, 1) = 1). Since R(4, 4) = 4, the root of this one-node optimal tree is its only key D. Figure 3.10 presents the optimal tree in its entirety.

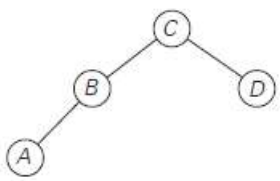


FIGURE 3.10 Optimal binary search tree for the above example.

3.4 KNAPSACK PROBLEM AND MEMORY FUNCTIONS

Designing a dynamic programming algorithm for the knapsack problem:

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Assume that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers.

0 / 1 knapsack problem means, the chosen item should be either null or whole.

Recurrence relation that expresses a solution to an instance of the knapsack problem

Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \le j \le W$. Let $F(i, j)$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do. Note the following:

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $F(i-1, j)$.
2. Among the subsets that do include the ith item (hence, $j - w_i \ge 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j-w_i)$.

Thus, the value of an optimal solution among all feasible subsets of the first I items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i-1$ items. These observations lead to the following recurrence:

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

It is convenient to define the initial conditions as follows:

$F(0, j) = 0$ for $j \ge 0$ and $F(i, 0) = 0$ for $i \ge 0$.

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

*For F(i, j),*compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left. The table can be filled either row by row or column by column.

ALGORITHM DPKnapsack($w[1..n]$, $v[1..n]$, W)

```
var V[0..n,0..W], P[1..n,1..W]: int
for j := 0 to W do
        V[0,j] := 0
for i := 0 to n do
        V[i,0] := 0
for i := 1 to n do
        for j := 1 to W do
```

if $w[i] \le j$ and $v[i] + V[i-1,j-w[i]] > V[i-1,j]$ then
$V[i,j] := v[i] + V[i-1,j-w[i]]$; $P[i,j] := j-w[i]$
else
$V[i,j] := V[i-1,j]$; $P[i,j] := j$
return $V[n,W]$ and the optimal subset by backtracing

Note: Running time and space: O(nW).

Table3.1 for solving the knapsack problem by dynamic programming.

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

EXAMPLE 1 Let us consider the instance given by the following data:
Table 3.2 An instance of the knapsack problem:

item	**weight**	**value**	capacity
1	2	$12	
2	1	$10	$W = 5$
3	3	$20	
4	2	$15	

The maximal value is $F(4, 5) = \$37$. We can find the composition of an optimal subset by **backtracing** (Back tracing finds the actual optimal subset, i.e. solution), the computations of this entry in the table. Since $F(4, 5) > F(3, 5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity. The value of the latter is $F(3, 3)$. Since $F(3, 3) = F(2, 3)$, item 3 need not be in an optimal subset. Since $F(2, 3) > F(1, 3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3 - 1)$ to specify its remaining composition. Similarly, since $F(1, 2) > F(0, 2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

Table 3.3 Solving an instance of the knapsack problem by the dynamic programming algorithm.

	Capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w1 = 2, v1 = 12$	1	0	0	12	12	12	12
$w2 = 1, v2 = 10$	2	0	10	12	22	22	22
$w3 = 3, v3 = 20$	3	0	10	12	22	30	32
$w4 = 2, v4 = 15$	4	0	10	15	25	30	**37**

Memory Functions

The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient.

The bottom up fills a table with solutions to all smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given.

Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using **memory functions.**

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm.

Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first: if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with i = n (the number of items) and j = W (the knapsack capacity).

ALGORITHM MFKnapsack(i, j)

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

EXAMPLE 2 Let us apply the memory function method to the instance considered in Example 1.

	Capacity j						
	I	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w1 = 2$, $v1 = 12$	1	0	0	12	12	12	12
$w2 = 1$, $v2 = 10$	2	0	-	12	22	-	22
$w3 = 3$, $v3 = 20$	3	0	-	-	22	-	32
$w4 = 2$, $v4 = 15$	4	0	-	-	-	-	37

Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, V (1, 2), is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger.

3.5 GREEDY TECHNIQUE

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step and this is the central point of this technique.

The choice made must be:

- *feasible*, i.e., it has to satisfy the problem's constraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm

Greedy Technique algorithms are:

- Prim's algorithm
- Kruskal's Algorithm
- Dijkstra's Algorithm
- Huffman Trees

Two classic algorithms for the minimum spanning tree problem: Prim's algorithm and Kruskal's algorithm. They solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimal solution.

Another classic algorithm named Dijkstra's algorithm used to find the shortest-path in a weighted graph problem solved by Greedy Technique . Huffman codes is an important data compression method that can be interpreted as an application of the greedy technique.

The first way is one of the common ways to do the proof for Greedy Technique is by **mathematical induction**.

The second way to prove optimality of a greedy algorithm is to show that on each step it does at least as well as any other algorithm could in **advancing** toward the problem's goal.

Example: find the minimum number of moves needed for a chess knight to go from one corner of a 100 × 100 board to the diagonally opposite corner. (The knight's moves are L-shaped jumps: two squares horizontally or vertically followed by one square in the perpendicular direction.)

A greedy solution is clear here: jump as close to the goal as possible on each move. Thus, if its start and finish squares are (1,1) and (100, 100), respectively, a sequence of 66 moves such as (1, 1) − (3, 2) − (4, 4) − . . . − (97, 97) − (99, 98) − (100, 100) solves the problem(The number k of two-move advances can be obtained from the equation 1+ 3k = 100).

Why is this a minimum-move solution? Because if we measure the distance to the goal by the Manhattan distance, which is the sum of the difference between the row numbers and the difference between the column numbers of two squares in question, the greedy algorithm decreases it by 3 on each move.

The third way is simply to show that the final result obtained by a greedy algorithm is optimal based on the **algorithm's output** rather than the way it operates.

Example: Consider the problem of placing the maximum number of chips on an 8 × 8 board so that no two chips are placed on the same or adjacent vertically, horizontally, or diagonally.

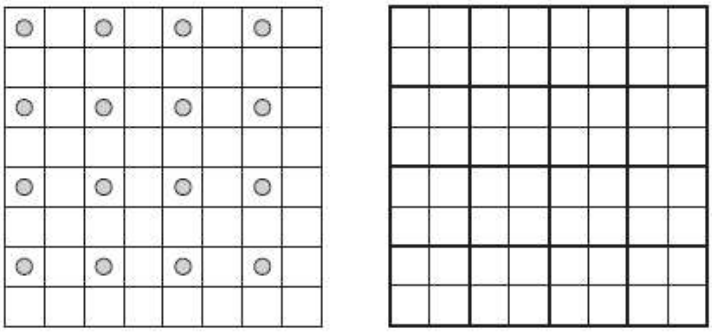


FIGURE 3.12 (a) Placement of 16 chips on non-adjacent squares. (b) Partition of the board proving impossibility of placing more than 16 chips.

It is impossible to place more than one chip in each of these squares, which implies that the total number of nonadjacent chips on the board cannot exceed 16.

3.6 PRIM'S ALGORITHM

A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

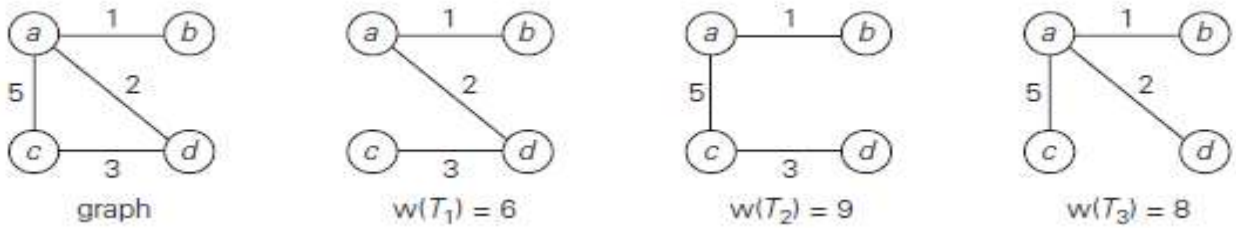


FIGURE 3.13 Graph and its spanning trees, with *T*1 being the minimum spanning tree.

The minimum spanning tree is illustrated in Figure 3. If we were to try constructing a minimum spanning tree by exhaustive search, we would face two serious obstacles. First, the number of spanning trees grows exponentially with the graph size (at least for dense graphs). Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a minimum spanning tree for a weighted graph.

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph $G = \{V, E\}$

//Output: E_T, the set of edges composing a minimum spanning tree of G

$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex

$E_T \leftarrow \Phi$

for $i \leftarrow 1$ **to** $|V| - 1$ **do**

find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)

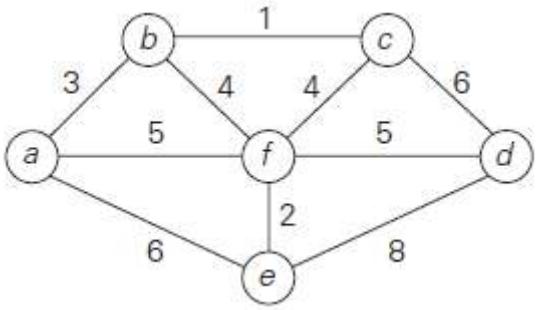
such that v is in V_T and u is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is $O(|E| \log |V|)$ in a connected graph, where $|V| - 1 \leq |E|$.



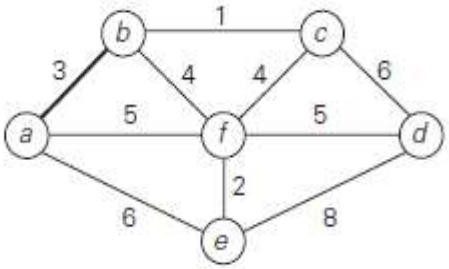
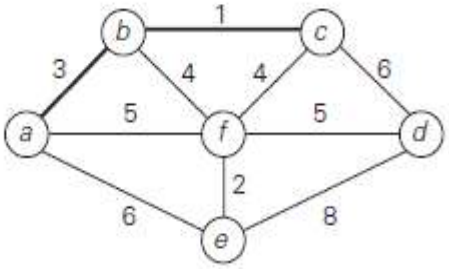
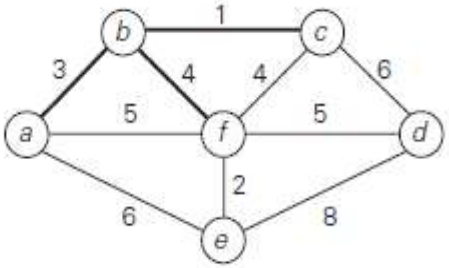
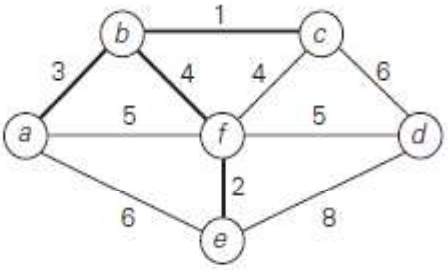
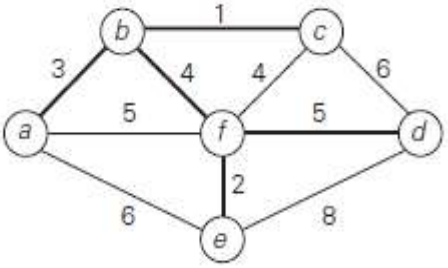
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

FIGURE 3.14 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are in bold.

3.7 KRUSKAL'S ALGORITHM

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G= {V, E} as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest. the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G = (V, E) as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest.

ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph $G = (V, E)$

//Output: E_T, the set of edges composing a minimum spanning tree of G

sort E in nondecreasing order of the edge weights $w(e_{i1}) \le \ldots \le w(e_{i\,|E|})$

$E_T \leftarrow \Phi$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size

$K \leftarrow 0$ //initialize the number of processed edges

while $ecounter < |V| - 1$ **do**

$k \leftarrow k + 1$

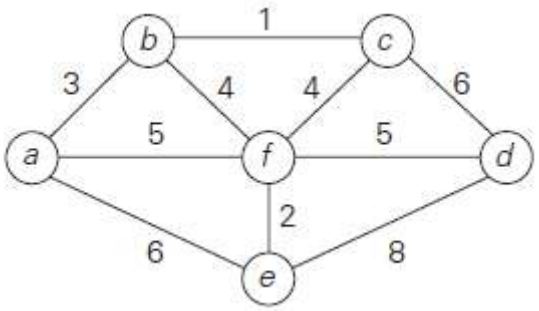
if $E_T \cup \{e_{ik}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{ik}\}$; $ecounter \leftarrow ecounter + 1$

return E_T

The initial forest consists of |V | trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called union-find algorithms. With an efficient union-find algorithm, the running time of Kruskal's algorithm will be $O(|E| \log |E|)$.



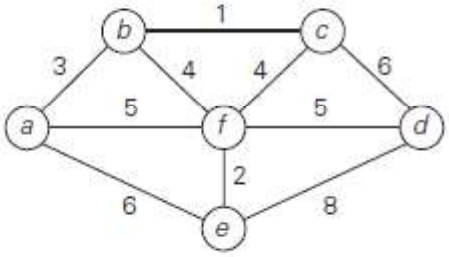
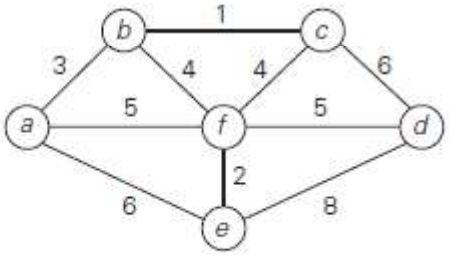
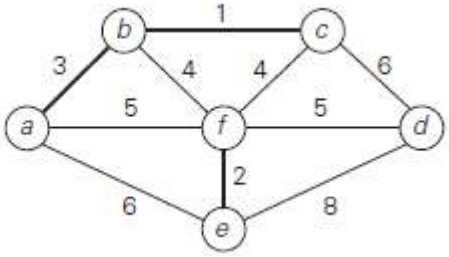
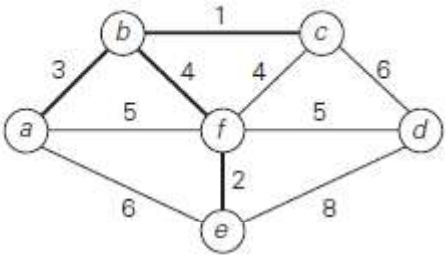
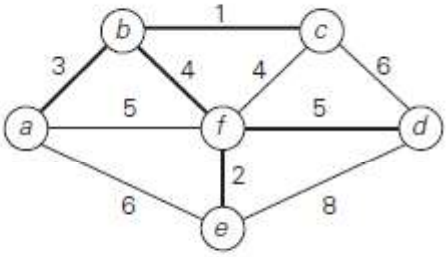
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

FIGURE 3.15 Application of Kruskal's algorithm. Selected edges are shown in bold.

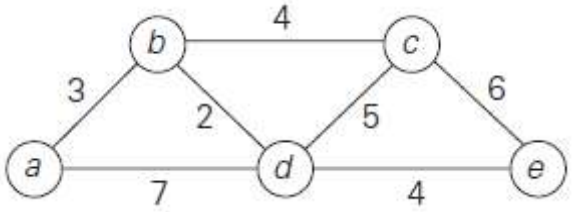
3.8 DIJKSTRA'S ALGORITHM

- Dijkstra's Algorithm solves the **single-source shortest-paths problem**.
- For a given vertex called the ***source*** in a weighted connected graph, find shortest paths to all its other vertices.
- The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have **edges in common**.
- The most widely used **applications** are transportation planning and packet routing in communication networks including the Internet.
- It also includes **finding shortest paths** in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling.
- In the world of **entertainment**, one can mention pathfinding in video games and finding best solutions to puzzles using their state-space graphs.
- Dijkstra's algorithm is the best-known algorithm for the single-source shortest-paths problem.

ALGORITHM *Dijkstra(G, s)*

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = (V, E)$ with nonnegative weights and its vertex s
//Output: The length dv of a shortest path from s to v and its penultimate vertex pv for every
// vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
    $dv \leftarrow \infty$; $pv \leftarrow$ **null**
    Insert (Q, v, dv) //initialize vertex priority in the priority queue
$Ds \leftarrow 0$; *Decrease(Q, s, d_s)* //update priority of s with d_s
$V_T \leftarrow \Phi$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
    $u^* \leftarrow$ *DeleteMin(Q)* //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $V - VT$ that is adjacent to u^* **do**
        if $d_{u^*} + w(u^*, u) < d_u$
        $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
        Decrease(Q, u, d_u)

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. It is in $\Theta(|V|^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in $O(|E| \log |V|)$.



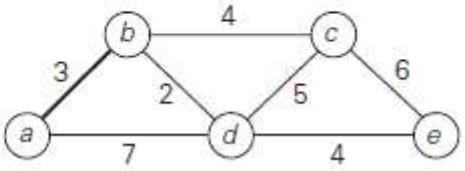
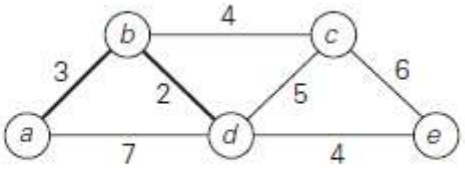
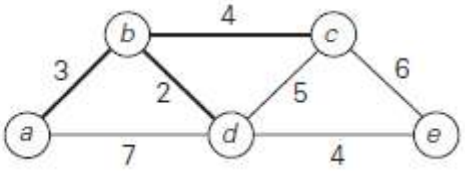
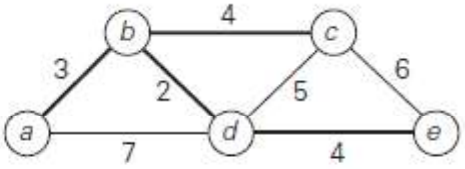
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

FIGURE 3.16 Application of Dijkstra's algorithm. The next closest vertex is shown in bold

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

From a to b : a − b of length 3

From a to d : a − b − d of length 5

From a to c : a − b − c of length 7

From a to e : a − b − d − e of length 9

3.9 HUFFMAN TREES

To encode a text that comprises symbols from some *n*-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the ***codeword***. For example, we can use a ***fixed-length encoding*** that assigns to each symbol a bit string of the same length m ($m \geq \log 2\ n$). This is exactly what the standard ASCII code does.

Variable-length encoding, which assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the *i*th) symbol? To avoid this complication, we can limit urselvesto the so-called ***prefix-free*** (or simply ***prefix***) ***codes***.

In a prefix code, no codeword is a prefix of a codeword of another symbol. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits by this symbol, and repeat this operation until the bit string's end is reached.

Huffman's algorithm

Step 1 Initialize *n* one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a **Huffman tree**. It defines in the manner described above is called a **Huffman code**.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 3.18

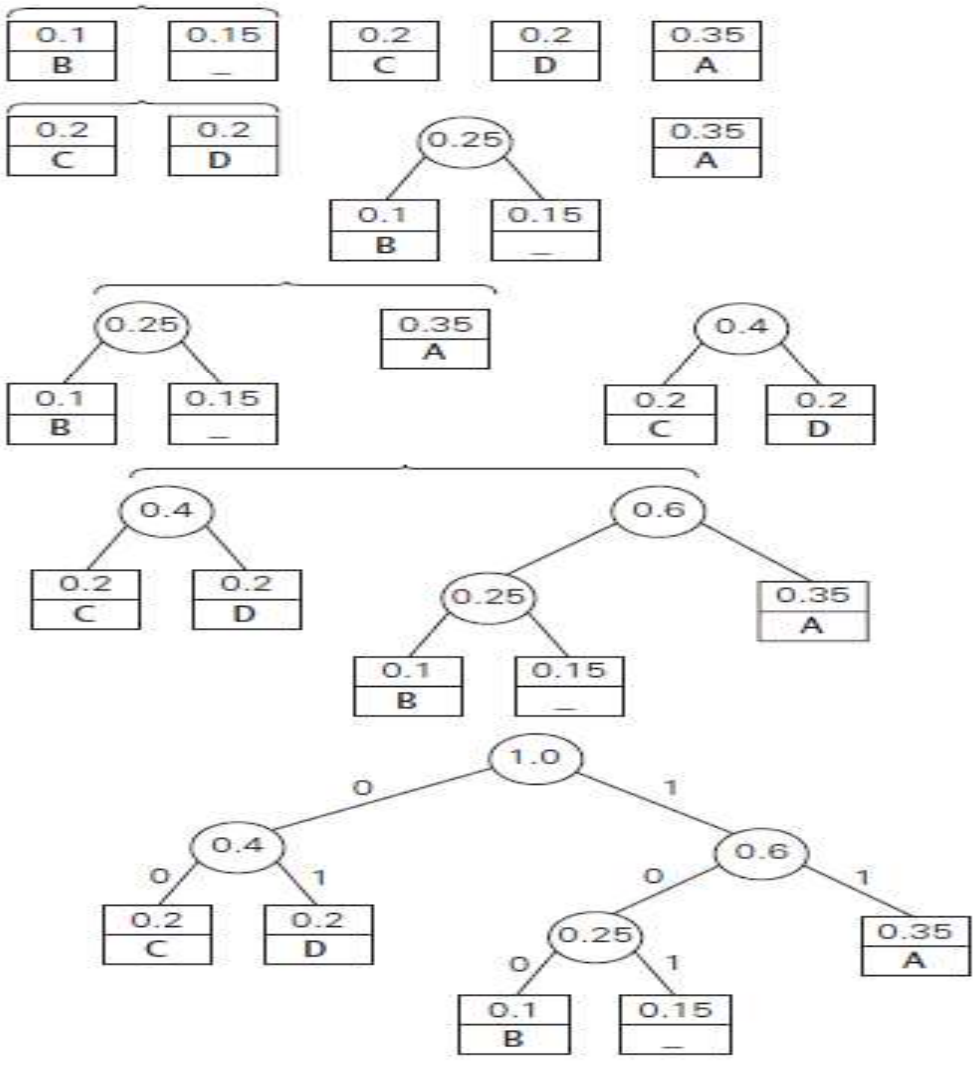


FIGURE 3.18 Example of constructing a Huffman coding tree.

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD. With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is 2 . 0.35 + 3 . 0.1+ 2 . 0.2 + 2 . 0.2 + 3 . 0.15 = 2.25.

We used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this toy example, Huffman's code achieves the ***compression ratio*** **-** a standard measure of a compression algorithm's effectiveness of (3− 2.25) / 3 · 100% = **25%**. In other words, Huffman's encoding of the text will use 25% less memory than its fixed-length encoding.

Running time is O(*n log n*), as each priority queue operation takes time O(*log n*).

Applications of Huffman's encoding

1. Huffman's encoding is a variable length encoding, so that number of bits used are lesser than fixed length encoding.
2. Huffman's encoding is very useful for file compression.
3. Huffman's code is used in transmission of data in an encoded format.
4. Huffman's encoding is used in decision trees and game playing.