4. FLOYD'S ALGORITHM

All-Pairs Shortest-Paths Problem:

Floyd's 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* × *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 $D^{(k)}$ $(i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n)$ 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 thank.

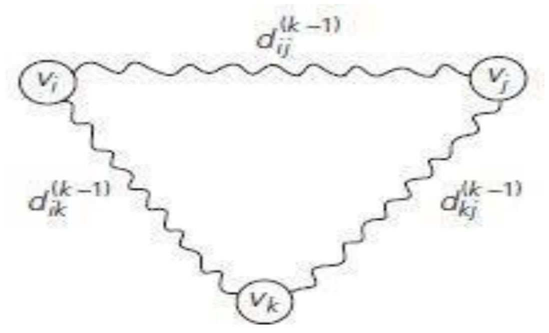


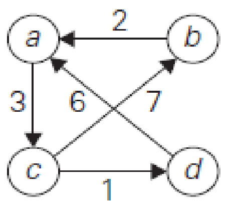
FIGURE 3.4 Underlying idea of Floyd's algorithm.

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

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

```
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.



$$D^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \infty & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \infty & 0 \end{array}$$

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

$$D^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & \mathbf{5} & \infty \\ c & \infty & 7 & 0 & 1 \\ d & 6 & \infty & \mathbf{9} & 0 \end{array}$$

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*).

$$D^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \infty & 3 & \infty \\ b & 2 & 0 & 5 & \infty \\ c & \mathbf{9} & 7 & 0 & 1 \\ d & 6 & \infty & 9 & 0 \end{array}$$

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*).

$$D^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & \mathbf{10} & 3 & \mathbf{4} \\ b & 2 & 0 & 5 & \mathbf{6} \\ c & 9 & 7 & 0 & 1 \\ d & 6 & \mathbf{16} & 9 & 0 \end{array}$$

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*).

$$D^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 10 & 3 & 4 \\ b & 2 & 0 & 5 & 6 \\ c & \mathbf{7} & 7 & 0 & 1 \\ d & 6 & 16 & 9 & 0 \end{array}$$

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.

5. MULTI STAGE GRAPH

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k* > 1**) number of disjoint subsets ***S* = {*s*₁,*s*₂,…,*s*ₖ}** such that edge *(u, v)* is in E, then $u \in s_i$ and $v \in s_{1+1}$ for some subsets in the partition and $|s_1| = |s_k| = 1$.

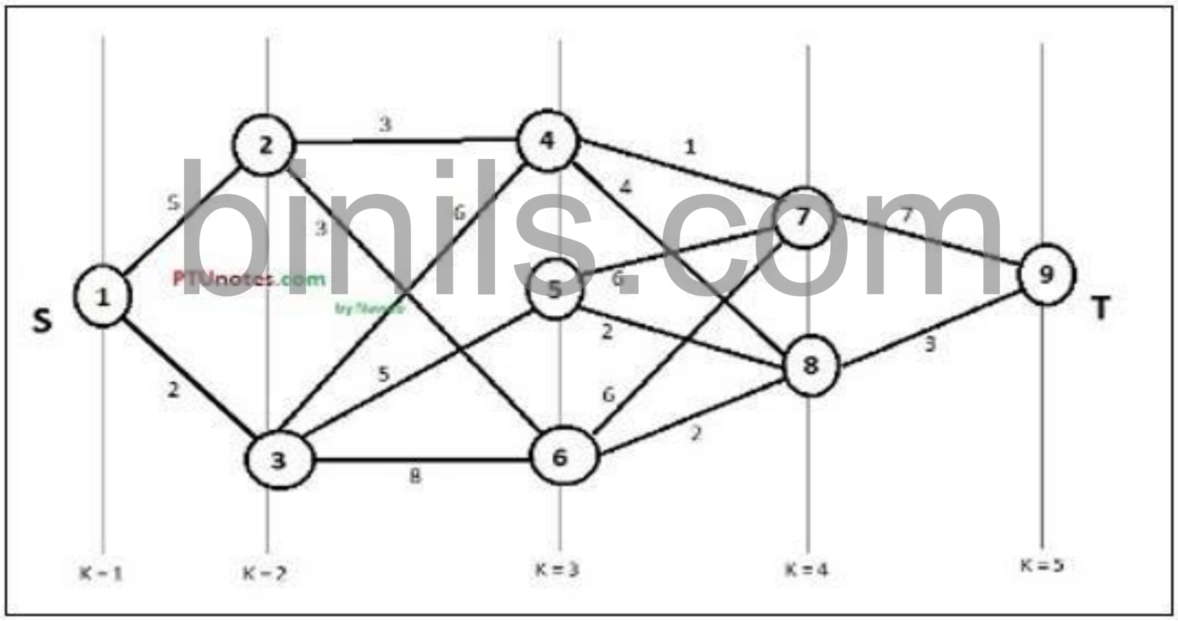
The vertex $\boldsymbol{s \in s_1}$ is called the **source** and the vertex $\boldsymbol{t \in s_k}$ is called **sink**.

G is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

Cost (3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost (3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost (3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost (2, 2) = min {c (2, 4) + Cost (4, 8) + Cost (8, 9), c (2, 6) +

Cost (6, 8) + Cost (8, 9)} = 8

Cost (2, 3) = {c (3, 4) + Cost (4, 8) + Cost (8, 9), c (3, 5) + Cost (5, 8) + Cost (8, 9), c (3, 6) + Cost (6, 8) + Cost (8, 9)} = 10

Step-3: Cost (K-4, j)

Cost (1, 1) = {c (1, 2) + Cost (2, 6) + Cost (6, 8) + Cost (8, 9), c (1, 3) + Cost (3, 5) + Cost (5, 8) + Cost (8, 9))} = 12

c (1, 3) + Cost (3, 6) + Cost (6, 8 + Cost (8, 9))} = 13

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

6. OPTIMAL BINARY SEARCH TREES

A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes. The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.

When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node

Search time of an element in a BST is ***O(n)***, whereas in a Balanced-BST search time is ***O(log n)***. Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

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.

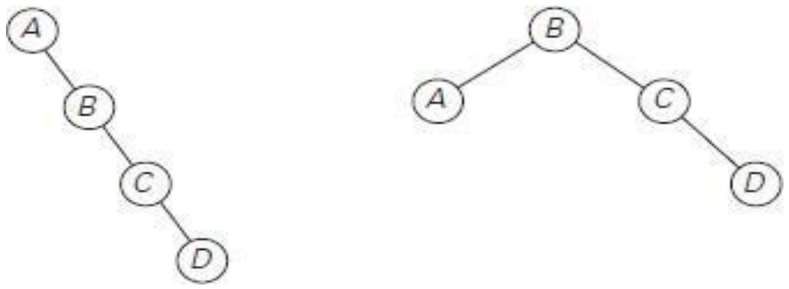


FIGURE 3.6.1 Two out of 14 possible binary search trees with keys *A, B, C,* and *D*.

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.

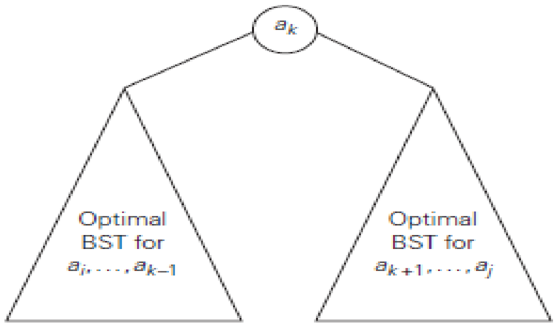
Neither of these two trees is optimal.

The total number of binary search trees with *n* keys is equal to the *n*th **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 largestandlet *p* $_1, \ldots, p_n$ be the probabilities of searching for them. Let *C(i, j)* be the smallest average


a_k
Optimal BST for $a_i, \ldots, a_{k-1}$
Optimal BST for $a_{k+1}, \ldots, a_j$

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 3.6.2 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.

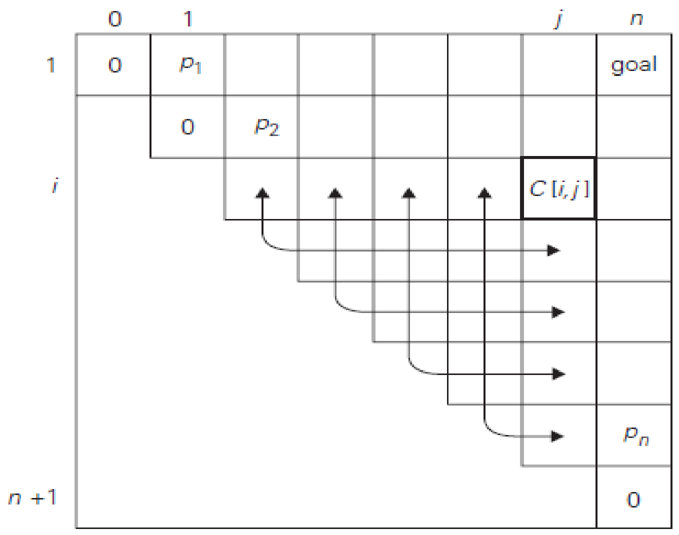


FIGURE 3.6.3 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

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.

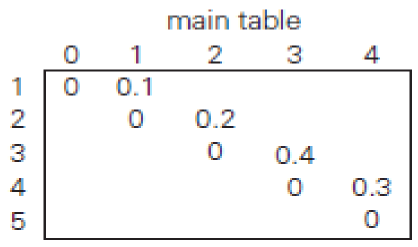
```
ALGORITHM OptimalBST(P [1..n])
    //Finds an optimal binary search tree by dynamic programming
    //Input: An array P[1..n] of search probabilities for a sorted list of n keys
    //Output: Average number of comparisons in successful searches in the
    // optimal BST and table R of subtrees' roots in the optimal BST
    for i ←1 to ndo
        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, **: Θ(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 **Θ(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]

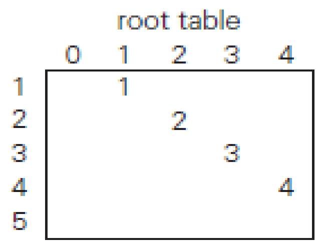
EXAMPLE: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	



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					

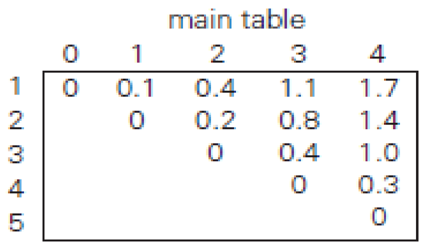
0.3 The initial tablesare:

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

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

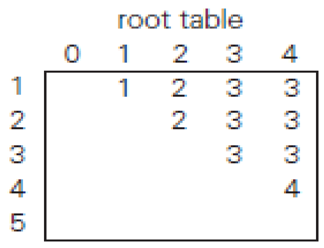
Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

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 treeisequalto1.7. 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).SinceR(4,4)=4,therootofthisone-nodeoptimaltreeisitsonlykeyD.Figure

3.10 presents the optimal tree in its entirety.

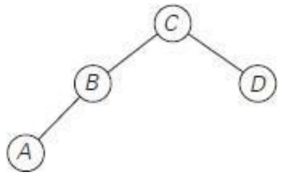


FIGURE 3.6.4 Optimal binary search tree for the above example.

7. KNAPSACK PROBLEM AND MEMORYFUNCTIONS

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 *i*th item and those that do. Note the following:

1. Among the subsets that do not include the *i*th item, the value of an optimal subset is, by definition, $F(i-1, j)$.
2. Among the subsets that do include the *i*th 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 $vi + F(i - 1, j - w_i)$.

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 *i*th 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:

$$F(0, j) = 0 \text{ for } j \geq 0 \text{ and } F(i, 0) = 0 \text{ for } i \geq 0.$$

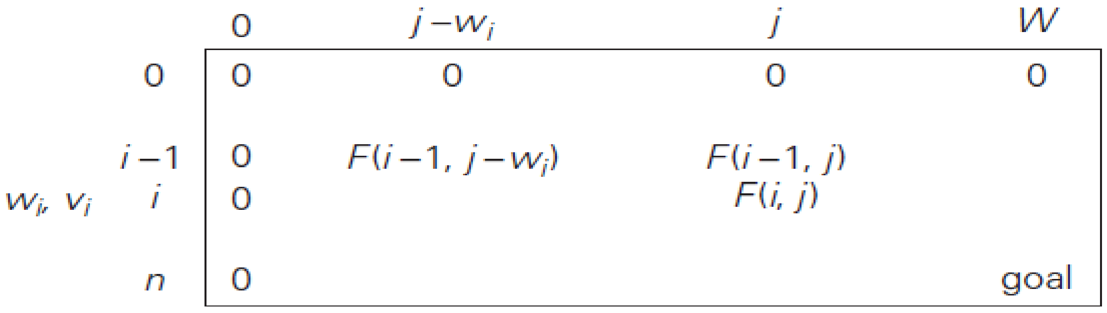
Our goal is to find *F(n, W)*, the maximal value of a subset of the *n* given items that fit into the knapsack of capacity *W*, and an optimal subset itself.

For F(i, j),compute the maximum of the entry in the previous row and the same column and the sum of *vi* 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] ≤ 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 back tracing
```

Note: Running timeandspace: O(nW).

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



		0	$j - w_i$	j	W
	0	0	0	0	0
	$i - 1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
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	$W = 5$
2	1	\$10	
3	3	\$20	
4	2	\$15	

The maximal value is $F(4, 5) = \$37$. We can find the composition of an optimal subset by **back tracing** (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
*w*1 = 2, *v*1 = 12	1	0	0	12	12	12	12
*w*2 = 1, *v*2 = 10	2	0	10	12	22	22	22
*w*3 = 3, *v*3 = 20	3	0	10	12	22	30	32
*w*4 = 2, *v*4 = 15	4	0	10	15	25	30	**37**

MEMORY FUNCTION:

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

The bottom up fills a table with solutions to all smaller sub problems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller sub problems 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 sub problems 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 with0'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

*w*4 = 2, *v*4 = 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.

8. CONTAINER LOADING PROBLEM

The greedy algorithm constructs the loading plan of a single container layer by layer from the bottom up. At the initial stage, the list of available surfaces contains only the initial surface of size *L* x *W* with its initial position at height 0. At each step, the algorithm picks the lowest usable surface and then determines the box type to be packed onto the surface, the number of the boxes and the rectangle area the boxes to be packed onto, by the procedure *select layer*.

The procedure *select layer* calculates a layer of boxes of the same type with the highest evaluation value. The procedure *select layer* uses breadth-limited tree search heuristic to determine the most promising layer, where the breadth is different depending on the different depth level in the tree search. The advantage is that the number of nodes expanded is polynomial to the maximal depth of the problem, instead of exponentially growing with regard to the problem size. After packing the specified number of boxes onto the surface according to the layer arrangement, the surface is divided into up to three sub-surfaces by the procedure *divide surfaces*.

Then, the original surface is deleted from the list of available surfaces and the newly generated sub-surfaces are inserted into the list. Then, the algorithm selects the new lowest usable surface and repeats the above procedures until no surface is available or all the boxes have been packed into the container. The algorithm follows a similar basic framework.

```
procedure greedy heuristic ()
list of surface: = initial surface of L x W at height 0 list of box type:= all box types
while (there exist usable surfaces) and (not all boxes are packed) do
 select the lowest usable surface as current surface set depth: = 0
 set best layer: = select layer (list of surface, list of box type, depth)
 pack best layer on current surface
 reduce the number of the packed box type by the packed amount
 Set a list of new surfaces: = divide surfaces (current surface, best layer, list of box type)
 delete current surface from the list of surfaces
 insert each surface in list of new surfaces into list of surfaces
end while
```

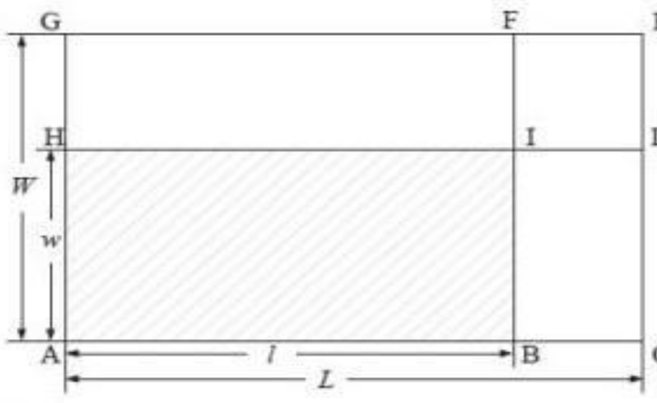


Fig: Division of the Loading Surface

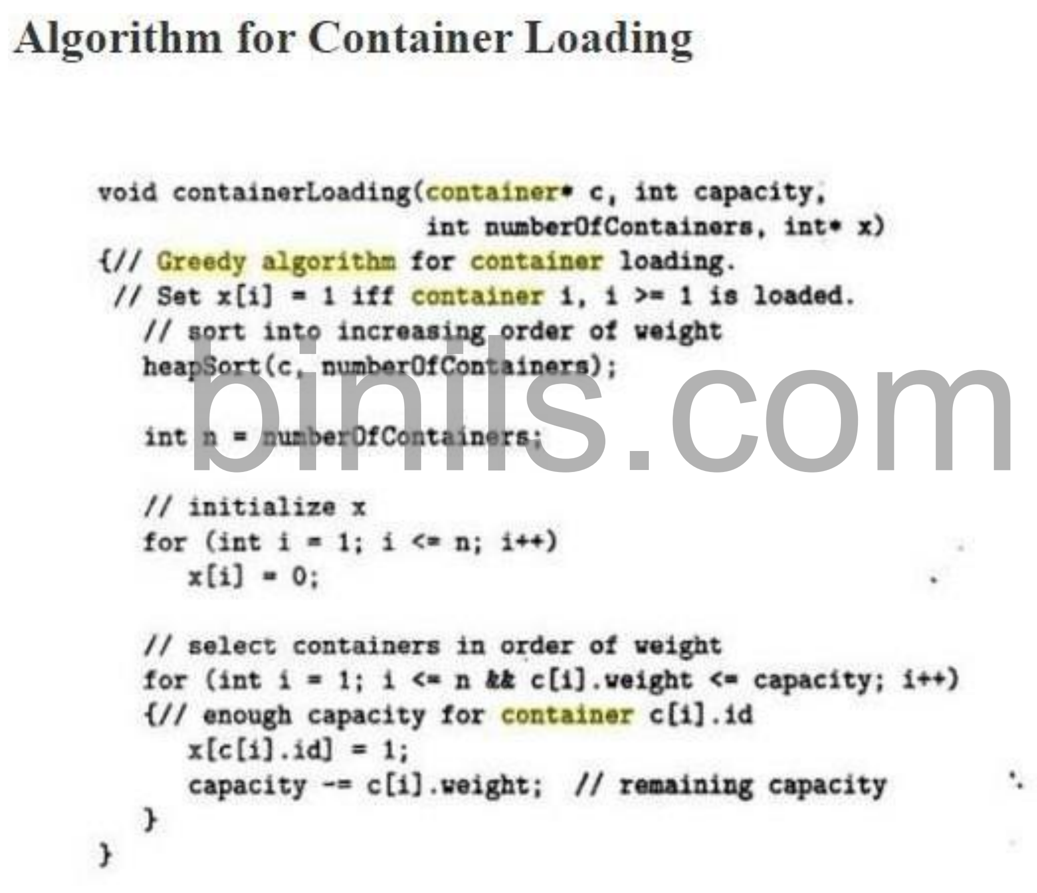
Given a layer of boxes of the same type arranged by the G4-heuristic, the layer is always packed in the bottom-left corner of the loading surface.

As illustrated in above Figure, up to three sub-surfaces are to be created from the original loading surface by the procedure *divide surfaces*, including the top surface, which is above the layer just packed, and the possible spaces that might be left at the sides.

If $l = L$ or $w = W$, the original surface is simply divided into one or two sub-surfaces, the top surface and a possible side surface. Otherwise, two possible division variants exist, i.e., to divide into the top surface, the surface (B,C,E,F) and the surface (F,G,H, I), or to divide into the top surface, the surface (B,C,D, I) and the surface (D,E,G,H).

The divisions are done according to the following criteria, which are similar to those in [2] and [5]. The primary criterion is to minimize the total unusable area of the division variant. If none of the remaining boxes can be packed onto a sub-surface, the area of the sub-surface is unusable. The secondary criterion is to avoid the creation of long narrow strips.

—The underlying rationale is that narrow areas might be difficult to fill subsequently‖. More specifically, if $L-l \geq W-w$, the loading surface is divided into the top surface, the surface (B,C,E,F) and the surface (F,G,H, I). Otherwise, it is divided into the top surface, the surface (B,C,D, I) and the surface (D,E,G,H).



Algorithm for Container Loading

```
void containerLoading(container* c, int capacity,
                      int numberOfContainers, int* x)
{// Greedy algorithm for container loading.
 // Set x[i] = 1 iff container i, i >= 1 is loaded.
   // sort into increasing order of weight
   heapSort(c, numberOfContainers);

   int n = numberOfContainers;

   // initialize x
   for (int i = 1; i <= n; i++)
      x[i] = 0;

   // select containers in order of weight
   for (int i = 1; i <= n && c[i].weight <= capacity; i++)
   {// enough capacity for container c[i].id
      x[c[i].id] = 1;
      capacity -= c[i].weight;  // remaining capacity
   }
}
```

9. 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.

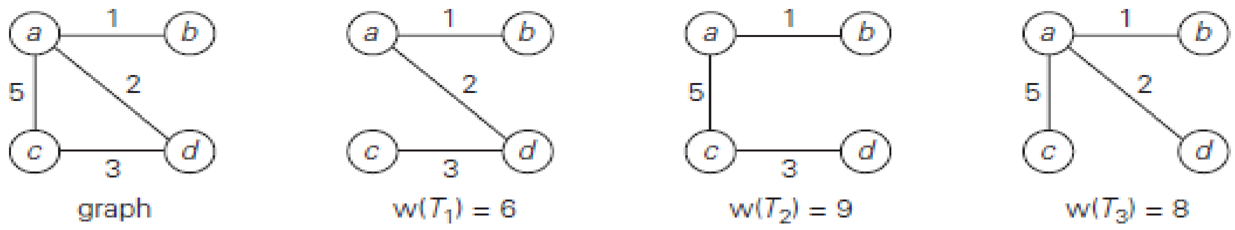


FIGURE 3.13 Graph and its spanning trees, with *T*1 being the minimum spanning tree.

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).

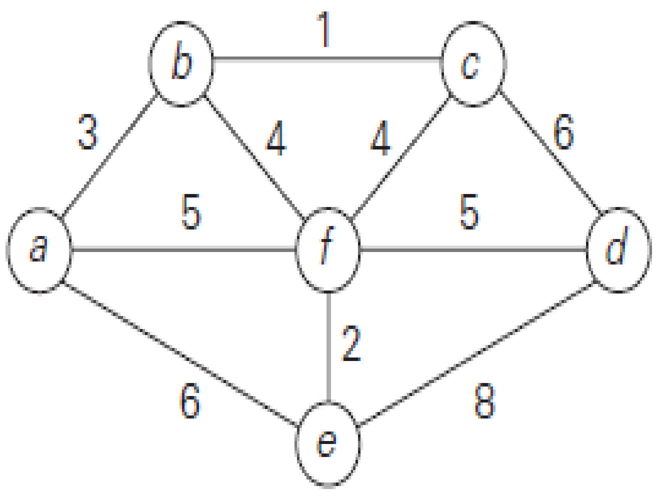
Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a minimum spanning tree for a weighted graph.

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 $|V| - 1 \leq |E|$.



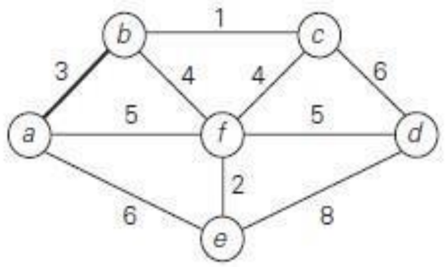
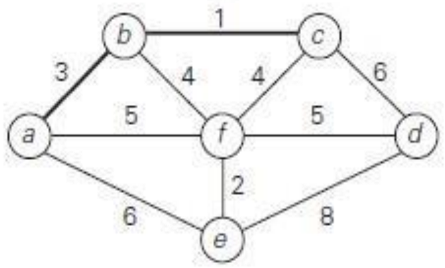
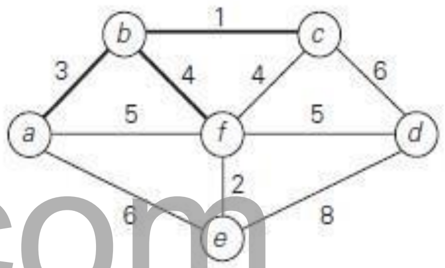
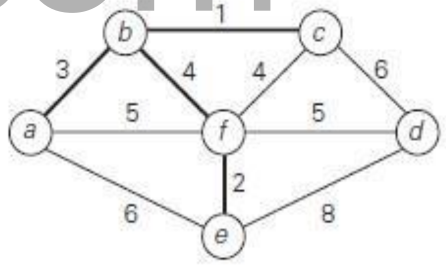
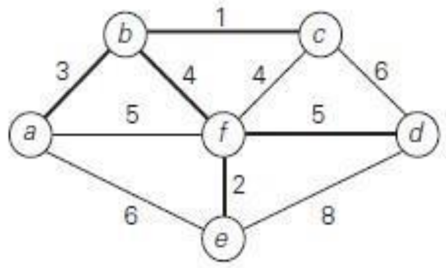
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

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.

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 no decreasing 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.

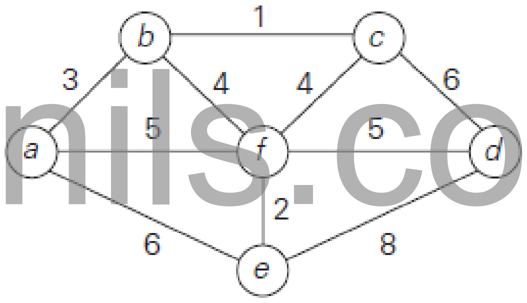
ALGORITHM *Kruskal(G)*

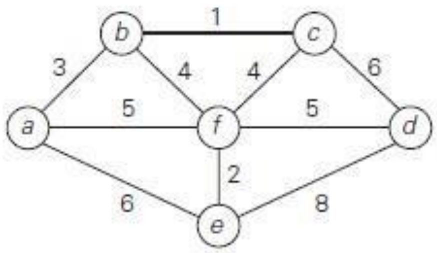
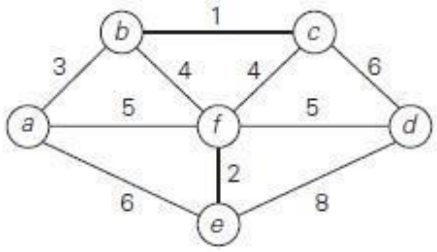
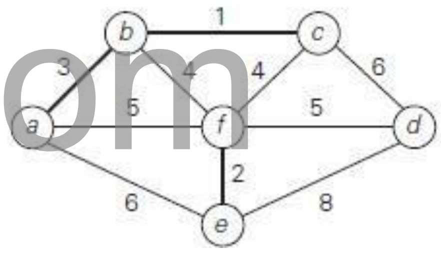
```
//Kruskal’s algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = (V, E)$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
        sort $E$ in nondecreasing order of the edge weights $w(e_{i1}) \le . .$
. $\le w(e_{i|E|})$ $E_T \leftarrow \Phi$; $ecounter \leftarrow 0$  //initialize the set of tree edges and
itssize

$K \leftarrow 0$                    //initialize the number of processededges
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	

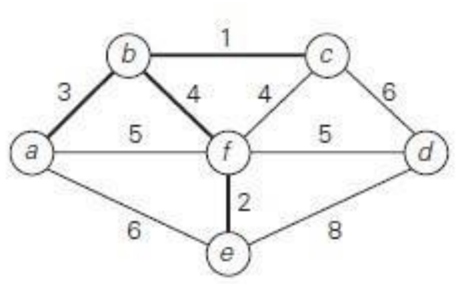
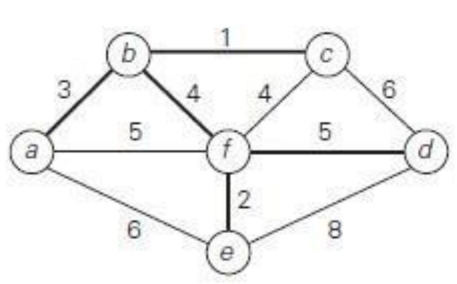
ab 3	bc 1 ef 2 ab 3 **bf 4** cf 4 af 5 df 5 ae 6 cd 6 de 8	
bf 4	bc 1 ef 2 ab 3 bf 4 cf 4 af 5 **df 5** ae 6 cd 6 de 8	
df 5		

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

DIJKSTRA'S ALGORITHM

- Dijkstra's Algorithm solves the **single-source shortest-pathsproblem**.
- For a given vertex called the ***source*** in a weighted connected graph, find shortest paths to all its othervertices.
- 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-spacegraphs.

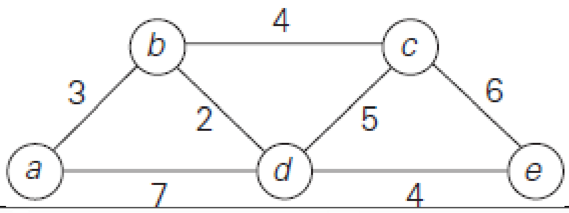
- Dijkstra's algorithm is the best-known algorithm for the single-source shortest-paths problem.

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 $O(|E| \log |V|)$.



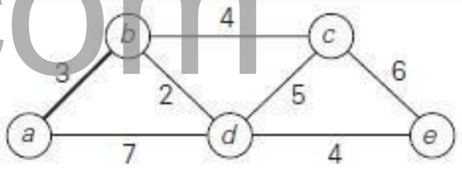
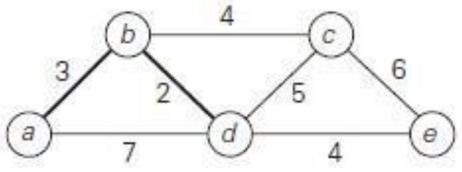
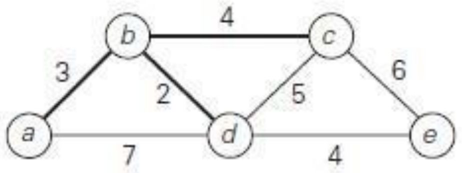
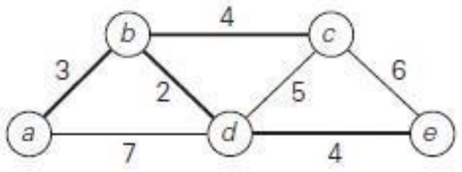
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

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

10. HUFFMANTREES

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 ***code word***. For example, we can use a ***fixed- length encoding*** that assigns to each symbol a bit string of the same length *m* (*m* ≥ log2 *n*).

This is exactly what the standard ASCII code does. ***Variable-length encoding***, which assigns code words 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 ourselves to the so-called ***prefix-free*** (or simply ***prefix***) ***codes***.

In a prefix code, no code word is a prefix of a code word 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 code word for some symbol, replace these bits by this symbol, and repeat this operation until the bit string's end is reached.

Huffman 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

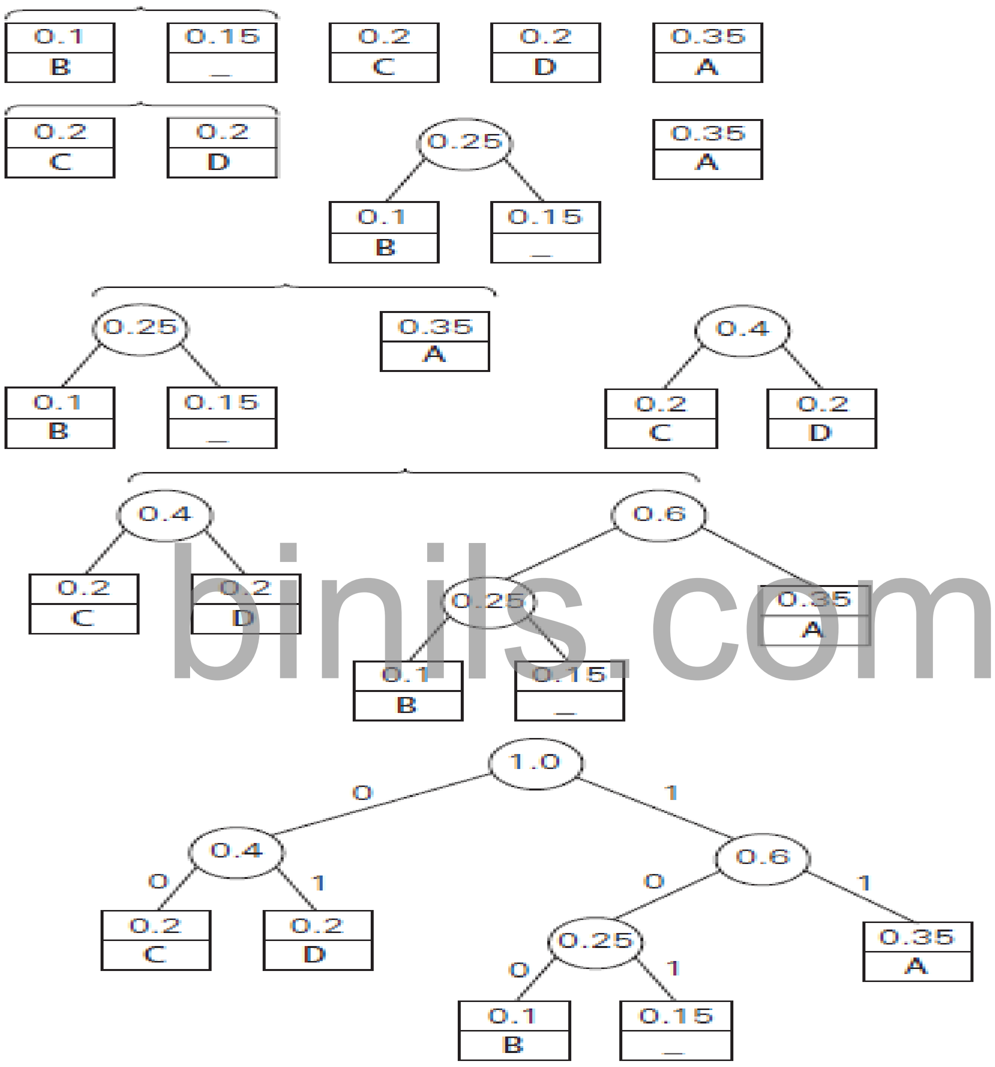


FIGURE 3.10.1 Example of constructing a Huffman coding tree.

The resulting code words are as follows:

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

Code word	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 code word 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.