UNIT 2 GREEDY TECHNIQUES

Structure **Page Nos.**

2.0 INTRODUCTION

Algorithms based on Greedy technique are used for solving **optimization problems**. **An optimization problem** is one in which some value (or set of values) of interest is required to be either minimized or maximized w.r.t some given relation on the values. Such problems include maximizing profits or minimizing costs of, say, production of some goods. Other examples of optimization problems are about

- finding the *minimum number* of currency notes required for an amount, say of Rs. 289, where arbitrary number of currency notes of each denomination from Rs. 1 to Rs. 100 are available, and
- finding shortest path covering a number of given cities where distances between pair of cities are given.

As we will study later, the algorithms based on greedy technique, *if exist*, are easy to think of, implement and explain about. However, for many interesting optimization problems, no algorithm based on greedy technique may yield optimal solution. In support of this claim, let us consider the following example:

Example 1.1: Let us suppose that we have to go from city A to city E through either city B or city C or city D with costs of reaching between pairs of cities as shown below:

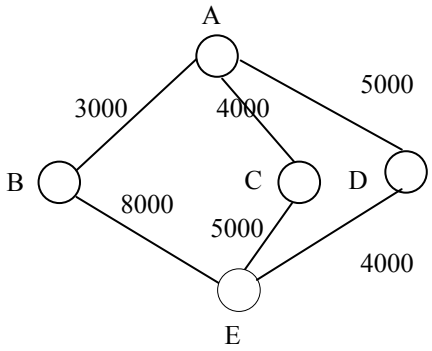


Figure 2.0.1

Then the greedy technique suggests that we take the route from A to B, the cost of which Rs.3000, is the minimum among the three costs, (viz., Rs. 3000, Rs. 4000 and Rs. 5000) of available routes.

However, at B there is only one route available to reach E. Thus, greedy algorithm suggests the route from A to B to E, which costs Rs.11000. But, the route from A to C to E, costs only Rs.9000. Also, the route from A to D to E costs also Rs.9000. Thus, locally better solution, at some stage, suggested by greedy technique yields overall (or globally) costly solution.

The essence of Greedy Technique is : In the process of solving an optimization problem, initially and at subsequent stages, we evaluate the costs/benefits of the various available alternatives for the next step. Choose the alternative which is *optimal* in the sense that either it is the least costly or it is the maximum profit yielding. In this context, it may be noted that the *overall solution*, yielded by choosing *locally optimal steps*, may *not* be optimal.

2.1 OBJECTIVES

After studying unit, you should be able to:

- explain the Greedy technique for solving optimization problems;
- apply the Greedy technique for solving optimization problems;
- apply Greedy technique for solving well-known problems including shortest path problem.

2.2 SOME EXAMPLES

In order to set the subject-matter to be discussed in proper context, we recall the general characteristics of greedy algorithms. These algorithms are:

- used to solve optimization problems,
- most straight forward to write,
- easy to invent, easy to implement, and if exist, are efficient,
- may not yield a solution for an arbitrary solvable problem,
- short-sighted making decisions on the basis of information immediately on hand, without worrying about the effect these decisions may have in future, and
- never reconsider their decisions.

In order to understand the salient features of the algorithms based on greedy technique, let us consider some examples in respect of the following **Minimum Number of Notes Problem:**

In a business transaction, we are required to make payment of some amount A (say Rs.289/-). We are given Indian currency notes of all denominations, viz., of 1,2,5,10,20,50,100, 500 and 1000. *The problem is to find the minimum number of currency notes to make the required amount A, for payment.* Further, it is assumed that currency notes of each denomination are available in sufficient numbers, so that one may choose as many notes of the same denomination as are required for the purpose of using the minimum number of notes to make the amount.

Example 2.2.1

In this example, we discuss, how intuitively we attempt to solve the Minimum Number of Notes Problem , to be specific, to make an amount of Rs.289/-.

Solution: Intuitively, to begin with, we pick up a note of denomination D, satisfying the conditions.

i) $D \leq 289$ and

ii) *if* D_1 is another denomination of a note such that $D_1 \leq 289$, *then* $D_1 \leq D$.

In other words, the picked-up note's denomination D is the largest among all the denominations satisfying condition (i) above.

The above-mentioned step of picking note of denomination D, satisfying the above two conditions, is repeated till either the amount of Rs.289/- is formed or we are clear that we can not make an amount or Rs.289/- out of the given denominations.

We apply the above-mentioned intuitive solution as follows:

To deliver Rs. 289 with minimum number of currency notes, the notes of different denominations are chosen and rejected as shown below:

Chosen-Note-Denomination	Total-Value-So far
100	0+100 ≤ 289
100	100+100= ≤ 289
~~100~~	~~200+100 > 289~~
50	200+50 ≤ 289
~~50~~	~~250+50 > 289~~
20	250 + 20 ≤ 289
~~20~~	~~270 + 20 > 289~~
10	270 + 10 ≤ 289
~~10~~	~~280 + 10 > 289~~
5	280 + 5 ≤ 289
~~5~~	~~285 + 5 > 289~~
2	285 + 2 < 289
2	287 + 2 = 289

The above sequence of steps based on Greedy technique, constitutes an algorithm to solve the problem.

To summarize, in the above mentioned solution, we have used the strategy of choosing, at any stage, the maximum denomination note, subject to the condition that the sum of the denominations of the chosen notes does not exceed the required amount A = 289.

The above strategy is the essence of greedy technique.

Example 2.2.2

Next, we consider an example in which for a given amount A and a set of available denominations, the **greedy algorithm does not provide a solution**, even when a solution by some other method exists.

Let us consider a hypothetical country in which notes available are of only the denominations 20, 30 and 50. We are required to collect an amount of 90.

Attempted solution through above-mentioned strategy of greedy technique:

i) First, pick up a note of denomination 50, because $50 \leq 90$. The amount obtained by adding denominations of all notes picked up so far is 50.

ii) Next, we can not pick up a note of denomination 50 again. However, if we pick up another note of denomination 50, then the amount of the picked-up

notes becomes 100, which is greater than 90. Therefore, we do not pick up any note of denomination 50 or above.

iii) Therefore, we pick up a note of next denomination, viz., of 30. The amount made up by the sum of the denominations 50 and 30 is 80, which is less then 90. Therefore, we accept a note of denomination 30.

iv) Again, we can not pick up another note of denomination 30, because otherwise the sum of denominations of picked up notes, becomes 80+30=110, which is more than 90. Therefore, we do not pick up only note of denomination 30 or above.

v) Next, we attempt to pick up a note of next denomination, viz., 20. But, in that case the sum of the denomination of the picked up notes becomes 80+20=100, which is again greater than 90. Therefore, we do not pick up only note of denomination 20 or above.

vi) Next, we attempt to pick up a note of still next lesser denomination. However, there are no more lesser denominations available.

Hence greedy algorithm fails to deliver a solution to the problem.

However, by some other technique, we have the following solution to the problem: First pick up a note of denomination 50 then two notes each of denomination 20.

Thus, we get 90 and , it can be easily seen that at least 3 notes are required to make an amount of 90. Another alternative solution is to pick up 3 notes each of denomination 30.

Example 2.2.3

Next, we consider an example, in which the greedy technique, of course, leads to a solution, but the solution yielded by **greedy technique is *not optimal***.

Again, we consider a hypothetical country in which notes available are of the only denominations 10, 40 and 60. We are required to collect an amount of 80.

Using the greedy technique, to make an amount of 80, first, we use a note of denomination 60. For the remaining amount of 20, we can choose note of only denomination 10. And , finally, for the remaining amount, we choose another note of denomination 10. **Thus, greedy technique suggests the following solution using 3 notes: 80 = 60 + 10 + 10.**

However, **the following solution uses only two notes:**

80 = 40 + 40

Thus, the solutions suggested by Greedy technique may not be optimal.

Ex.1) Give another example in which greedy technique fails to deliver an optimal solution.

2.3 FORMALIZATION OF GREEDY TECHNIQUE

In order to develop an *algorithm* based on *the greedy technique* to solve a *general optimization problem*, we need the following data structures and functions:

(i) **A set or list of given/candidate values** from which choices are made, to reach a solution. For example, in the case of Minimum Number of Notes problem, the list of candidate values (in rupees) of notes is {1, 2, 5, 10, 20, 50, 100, 500, 1000}. Further, the number of notes of each denomination should be clearly

mentioned. Otherwise, it is assumed that each candidate value can be used as many times as required for the solution using greedy technique. Let us call this set as

GV: Set of Given Values

(ii) **Set (*rather multi-set*) of considered and chosen values:** This structure contains those candidate values, which are considered and chosen by the algorithm based on greedy technique to reach a solution. Let us call this structure as

CV: Structure of Chosen Values

The structure is generally not a set but a multi-set in the sense that values may be repeated. For example, in the case of Minimum Number of Notes problem, if the amount to be collected is Rs. 289 then

CV = {100, 100, 50, 20, 10, 5, 2, 2}

(iii) **Set of Considered and Rejected Values:** As the name suggests, this is the set of all those values, which are considered but rejected. Let us call this set as

RV: Set of considered and Rejected Values

A candidate value may belong to both CV and RV. But, once a value is put in RV, then this value can not be put any more in CV. For example, to make an amount of Rs. 289, once we have chosen two notes each of denomination 100, we have

CV = {100, 100}

At this stage, we have collected Rs. 200 out of the required Rs. 289. At this stage RV = {1000, 500}. So, we can chose a note of any denomination except those in RV, i.e., except 1000 and 500. Thus, at this stage, we can chose a note of denomination 100. However, this choice of 100 again will make the total amount collected so far, as Rs. 300, which exceeds Rs. 289. Hence we reject the choice of 100 third time and put 100 in RV, so that now RV = {1000, 500, 100}. From this point onward, we can not chose even denomination 100.

Next, we consider some of the functions, which need to be defined in an algorithm using greedy technique to solve an optimization problem.

(iv) **A function say *SolF*** that checks whether a solution is reached or not. However, the function does not check for the *optimality* of the obtained solution. In the case of Minimum Number of Notes problem, the function SolF finds the sum of all values in the multi-set CV and compares with the desired amount, say Rs. 289. For example, if at one stage CV = {100, 100} then sum of values in CV is 200 which does not equal 289, then the function SolF returns '*Solution not reached*'. However, at a later stage, when CV = {100, 100, 50, 20, 10, 5, 2, 2}, then as the sum of values in CV equals the required amount, hence the function SolF returns the message of the form '*Solution reached*'.

It may be noted that the function only informs about a possible solution. However, solution provided through SolF may not be *optimal*. For instance in the Example 2.2.3, when we reach CV = {60, 10, 10}, then SolF returns '*Solution reached*'. However, as discussed earlier, the solution 80 = 60 + 10 + 10 using three notes is not optimal, because, another solution using only two notes, viz., 80 = 40 + 40, is still cheaper.

(v) **Selection Function** say ***SelF*** finds out the most promising candidate value out of the values not yet rejected, i.e., which are not in RV. In the case of Minimum Number of Notes problem, for collecting Rs. 289, at the stage when RV = {1000, 500} and CV = {100, 100} then first the function SelF attempts the denomination 100. But, through function SolF, when it is found that by

addition of 100 to the values already in CV, the total value becomes 300 which exceeds 289, the value 100 is rejected and put in RV. Next, the function SelF attempts the next lower denomination 50. The value 50 when added to the sum of values in CV gives 250, which is less than 289. Hence, the value 50 is returned by the function SelF.

(vi) **The Feasibility-Test Function,** say **FeaF.** When a new value say v is chosen by the function SelF, then the function FeaF checks whether the new set, obtained by adding v to the set CV of already selected values, is a possible part of the final solution. Thus in the case of Minimum Number of Notes problem, if amount to be collected is Rs. 289 and at some stage, CV = {100, 100}, then the function SelF returns 50. At this stage, the function FeaF takes the control. It adds 50 to the sum of the values in CV, and on finding that the sum 250 is less than the required value 289 informs the main/calling program that {100, 100, 50} can be a part of some final solution, and needs to be explored further.

(vii) **The Objective Function,** say **ObjF,** gives the value of the solution. For example, in the case of the problem of collecting Rs. 289; as CV = {100, 100, 50, 20, 10, 5, 2, 2} is such that sum of values in CV equals the required value 289, the function ObjF **returns the number** of notes in CV, i.e., the number 8.

After having introduced a number of sets and functions that may be required by an algorithm based on greedy technique, we give below the outline of greedy technique, say **Greedy-Structure.** For any *actual* algorithm based on greedy technique, the various *structures* the functions discussed above have to be replaced by *actual* functions.

These functions depend upon the problem under consideration. The Greedy-Structure outlined below takes the set GV of given values as input parameter and returns CV, the set of chosen values. For developing any algorithm based on greedy technique, the following function outline will be used.

2.3.1 Function Greedy-Structure (GV:set): set

```
    CV ← ϕ  {initially, the set of considered values is empty}
While GV ≠ RV and not SolF (CV) do
    begin
        v ← SelF (GV)
        If FeaF (CV ∪ {v})  then
            CV  ←  CV ∪ {v}
        else  RV ← RV ∪ {v}
    end

        // the function Greedy Structure comes out
    // of while-loop when either GV=RV, i.e., all
// given values are rejected or when solution is found

        If  SolF (CV) then returns ObjF (GV)
          else return "No solution is possible"
    end function Greedy–Structure
```

2.4 MINIMUM SPANNING TREE

In this section and some of the next sections, we apply greedy technique to develop algorithms to solve some well-known problems. First of all, we discuss the applications for finding minimum spanning tree for a given (undirected) graph. In order to introduce the subject matter, let us consider some of the relevant definitions.

Definitions

A ***Spanning tree*** of a connected graph, say G = (V, E) with V as set of vertices and E as set of edges, is its **connected** acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.

A ***minimum spanning tree*** of a *weighted* connected graph 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.

The minimum spanning tree problem has a number of useful applications in the following type of situations:

Suppose, we are given a set of cities alongwith the distances between each pair of cities. In view of the shortage of funds, it is desired that in stead of connecting directly each pair of cities, we provide roads, costing least, but allowing passage between any pair cities along the provided roads. However, the road between some pair of cities may not be direct, but may be passing through a number of other cities.

Next, we illustrate the concept of spanning tree and minimum spanning tree through the following example.

Let us consider the connected weighted graph G given in *Figure 4.1*.

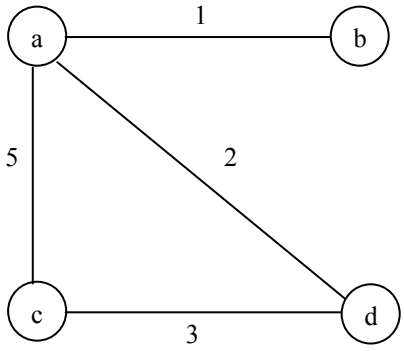


Figure: 2.4.1

For the graph of *Figure 2.4.1*given above, each of *Figure 2.4.2, Figure, 2. 4.3* and *Figure. 2.4.4* shows a spanning tree viz., T_1, T_2 and T_3 respectively.
Out of these, T_1 is a minimal spanning tree of G, of weight 1+2+3 = 6.

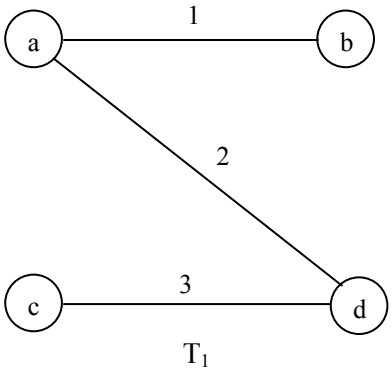
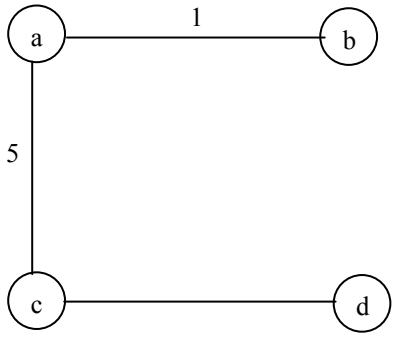
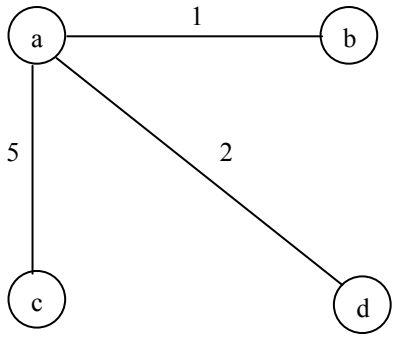


Figure: 2.4.2



T_2

Figure: 2.4.3



T_3

Figure: 2.4.4

Remark 2.4.1:

The weight may denote (i) length of an edge between pair of vertices or (ii) the cost of reaching from one town to the other or (iii) the cost of production incurred in reaching from one stage of production to the immediate next stage of production or (iv) the cost of construction of a part of a road or of laying telephone lines between a pair of towns.

Remark 2.4.2:

The weights on edges are generally positive. However, in some situations the weight of an edge may be zero or even negative. The negative weight *may appear* to be appropriate when the problem under consideration is not about '*minimizing costs*' but about '*maximizing profits'* and we still want to use *minimum spanning tree algorithms*. However, in such cases, it is not appropriate to use negative weights, because, more we traverse the negative-weight edge, lesser the cost. However, with repeated traversals of edges, the cost should increase in stead of decreasing.

Therefore, if we want to apply minimum spanning tree technique to 'maximizing profit problems, then in stead of using negative weights, we replace profits p_i by $M-p_i$ where M is some positive number s.t

$M > \text{Max}\ \{p_{ij} : p_{ij}$ is the profit in traversing from ith node to jth node$\}$

Remark 2.4.3:

From graph theory, we know that a given connected graph with n vertices, must have exactly (n –1) edges.

As mentioned earlier, whenever we want to develop an algorithm based on greedy technique, we use the function Greedy-Structure given under 2.3.1. For this purpose,

we need to find appropriate values of the various sets and functions discussed in Section 3.

In the case of **the problem of finding minimum-spanning tree for a given connected graph,** the appropriate values are as follows:

(i) **GV: The set of candidate or given values** is given by GV = E, the set of edges of the given graph (V, E).

(ii) **CV: The structure of chosen values** is given by those edges from E, which together will form the required minimum-weight spanning tree.

(iii) **RV: set of rejected values** will be given by those edges in E, which at some stage will form a cycle with earlier selected edges.

(iv) **In the case of the problem of minimum spanning tree, the function SolF that checks whether a solution** is reached or not, is the function that checks that

(a) all the edges in CV form a tree,
(b) the set of vertices of the edges in CV equal V, the set of all edges in the graph, and
(c) the sum of the weights of the edges in CV is minimum possible of the edges which satisfy (a) and (b) above.

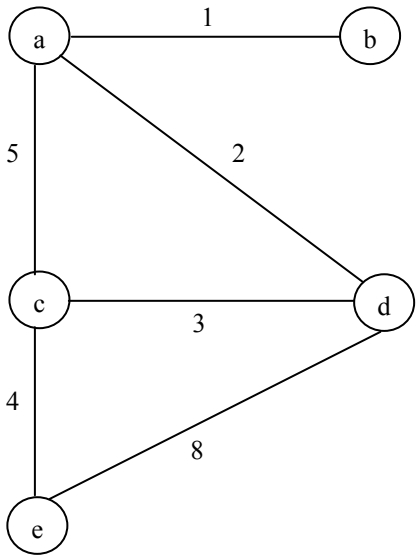
(v) **Selection Function:** depends upon the particular algorithm used for the purpose. There are two well-known algorithms, viz., Prim's algorithm and Kruskal's algorithm for finding the Minimum Spanning Tree. We will discuss these algorithms in detail in subsequent sections.

(vi) **FeaF: Feasibility Test Function:** In this case, when the selection function SelF returns an edge depending on the algorithm, the feasibility test function FeaF will check whether the newly found edge forms a cycle with the earlier slected edges. If the new edge actually forms a cycle then generally the newly found edge is dropped and search for still another edge starts. However, in some of the algorithms, it may happen that some earlier chosen edge is dropped.

(vii) In the case of Minimum Spanning Tree problem, the objective function may return

(a) the set of edges that constitute the required minimum spanning tree and
(b) the weight of the tree selected in (a) above.

Ex. 2) Find a minimal spanning tree for the following graph.



2.5 PRIM'S ALGORITHM

The algorithm due to **Prim** builds up a minimum spanning tree by adding edges to form a sequence of expanding subtrees. The sequence of subtrees is represented by the pair (V_T, E_T), where V_T and E_T respectively represent the set of vertices and the set of edges of a subtree in the sequence. Initially, the subtree, in the sequence, consists of just a single vertex which is selected arbitrarily from the set *V* of vertices of the given graph. The subtree is built-up iteratively by adding an edge that has minimum weight among the remaining edges (i.e., edge selected greedily) and, which at the same time, does not form a cycle with the earlier selected edges.

We illustrate the Prim's algorithm through an example before giving a semi-formal definition of the algorithm.

Example 2.5.1 (of Prim's Algorithm):

Let us explain through the following example how Prim's algorithm finds a minimal spanning tree of a given graph. Let us consider the following graph:

Initially

$V_T = (a)$

$E_T = \phi$

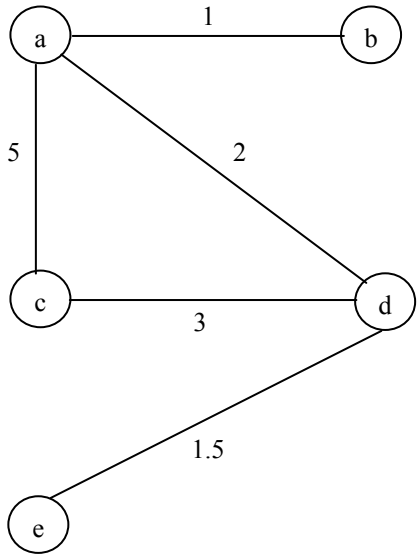


Figure: 2.5.1

In the first iteration, the edge having weight which is the minimum of the weights of the edges having **a** as one of its vertices, is chosen. In this case, the edge ab with weight 1 is chosen out of the edges **ab**, ac and ad of weights respectively 1,5 and 2. Thus, after First iteration, we have the **given graph with chosen edges in bold and V_T and E_T as follows:**

$V_T = (a, b)$

$E_T = (\ (a,b))$

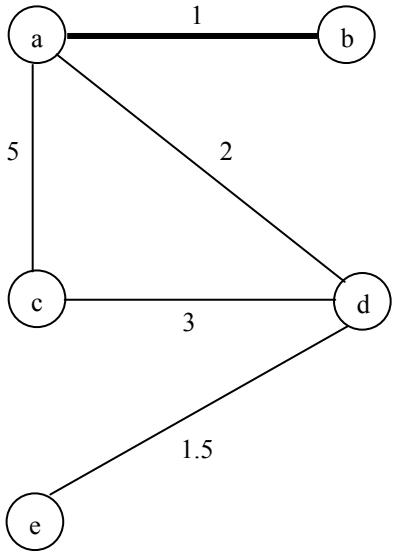


Figure: 2.5.2

In the next iteration, out of the edges, not chosen earlier and not making a cycle with earlier chosen edge and having either a or b as one of its vertices, the edge with minimum weight is chosen. In this case the vertex b does not have any edge originating out of it. In such cases, if required, weight of a non-existent edge may be taken as ∞. Thus choice is restricted to two edges viz., ad and ac respectively of weights 2 and 5. Hence, in the next iteration the edge ad is chosen. **Hence, after second iteration, we have the given graph with chosen edges and V_T and E_T as follows:**

V_T = (a, b, d)
E_T = ((a, b), (a, d))

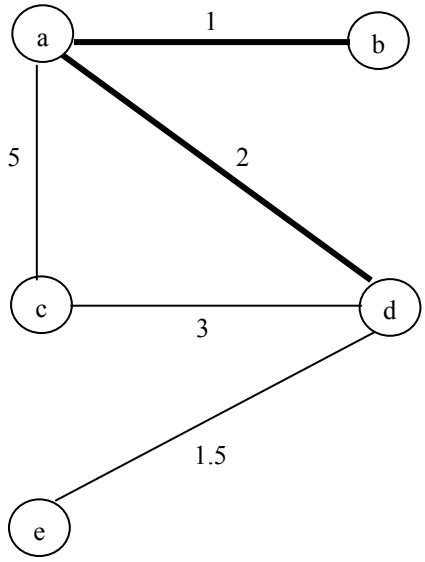


Figure: 2.5.3

In the next iteration, out of the edges, not chosen earlier and not making a cycle with earlier chosen edges and having either a, b or d as one of its vertices, the edge with minimum weight is chosen. Thus choice is restricted to edges ac, dc and de with weights respectively 5, 3, 1.5. The edge de with weight 1.5 is selected. **Hence, after third iteration we have the given graph with chosen edges and V_T and E_T as follows:**

V_T = (a, b, d, e)
E_T = ((a, b), (a, d), (d, e))

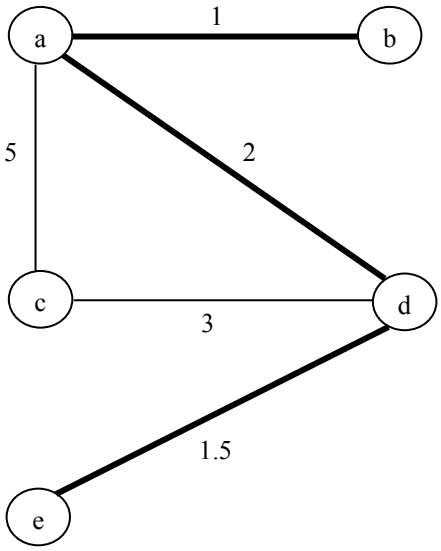


Figure: 2.5.4

In the next iteration, out of the edges, not chosen earlier and not making a cycle with earlier chosen edge and having either a, b, d or e as one of its vertices, the edge with minimum weight is chosen. Thus, choice is restricted to edges dc and ac with weights respectively 3 and 5. Hence the edge dc with weight 3 is chosen. Thus, after fourth iteration, we have **the given graph with chosen edges and V_T and E_T as follows:**

V_T = (a, b, d, e, c)
E_T = ((a b), (a d) (d e) (d c))

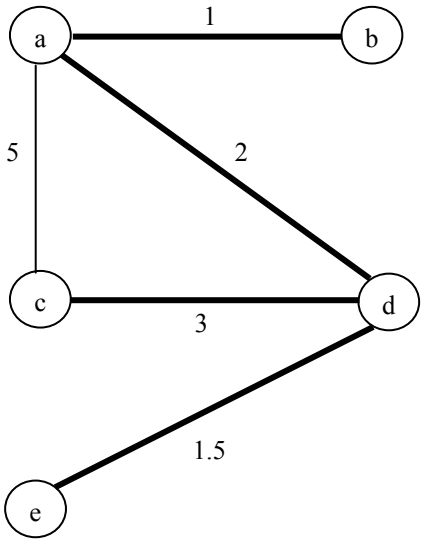


Figure: 2.5.5

At this stage, it can be easily seen that each of the vertices, is on some chosen edge and the chosen edges form a tree.

Given below is the semiformal definition of Prim's Algorithm

Algorithm Spanning-Prim (G)

// the algorithm constructs a minimum spanning tree
// for which the input is a weighted connected graph $G = (V, E)$
// the output is the set of edges, to be denoted by E_T, which together constitute a minimum spanning tree of the given graph G
// for the pair of vertices that are not adjacent in the graph to each other, can be given
// the label ∞ indicating "infinite" distance between the pair of vertices.
// the set of vertices of the required tree is initialized with the vertex v_0
$V_T \leftarrow \{v_O\}$
$E_T \leftarrow \phi$ // initially E_T is empty
// let n = number of vertices in V

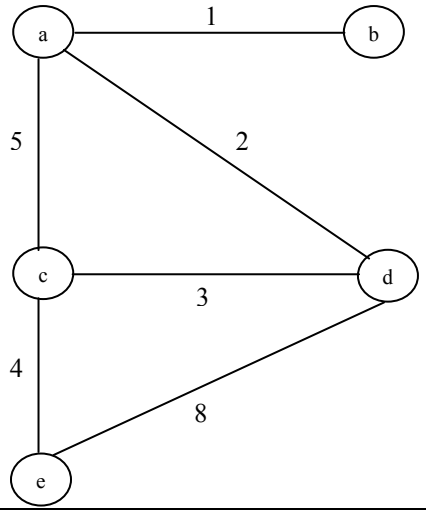
For i = 1 **to** |n| - 1 **do**

find a minimum-weight edge $\underline{e} = (v^1, u^1)$ among all the edges such that v^1 is in V_T and u^1 is in $V - V_T$.

$V_T \leftarrow V_T \cup \{u^1\}$
$E_T = E_T \cup \{\underline{e}\}$

Return E_T

Ex. 3) Using Prim's algorithm, find a minimal spanning tree for the graph given below:



2.6 KRUSKAL'S ALGORITHM

Next, we discuss another method, of finding minimal spanning tree of a given weighted graph, which is suggested by Kruskal. In this method, the emphasis is on the choice of edges of minimum weight from amongst all the available edges, of course, subject to the condition that chosen edges do not form a cycle.

The connectivity of the chosen edges, at any stage, in the form of a subtree, which was emphasized in Prim's algorithm, is **not** essential.

We briefly describe the Kruskal's algorithm to find minimal spanning tree of a given weighted and connected graph, as follows:

(i) First of all, order all the weights of the edges in increasing order. Then repeat the following two steps till a set of edges is selected containing all the vertices of the given graph.

(ii) Choose an edge having the weight which is the minimum of the weights of the edges not selected so far.

(iii) If the new edge forms a cycle with any subset of the earlier selected edges, then drop it, *else,* add the edge to the set of selected edges.

We illustrate the Kruskal's algorithm through the following:

Example 2.6.1:

Let us consider the following graph, for which the minimal spanning tree is required.

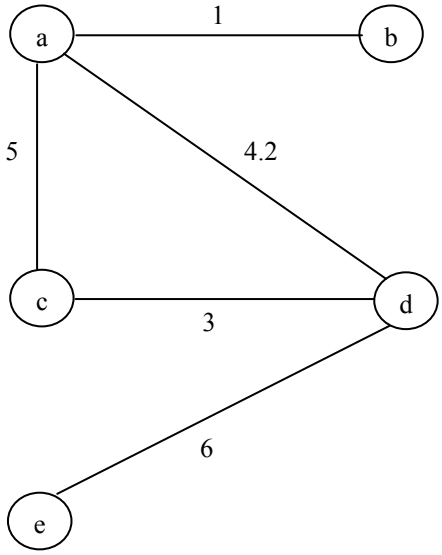


Figure: 2.6.1

Let E_g denote the set of edges of the graph that are chosen upto some stage.

According to the step (i) above, the weights of the edges are arranged in increasing order as the set

{1, 3, 4.2 ,5, 6}

In the first iteration, the edge (a,b) is chosen which is of weight 1, the minimum of all the weights of the edges of the graph.

As single edge do not form a cycle, therefore, the edge (a,b) is selected, so that

E_g = ((a,b))

After first iteration, the graph with selected edges in bold is as shown below:

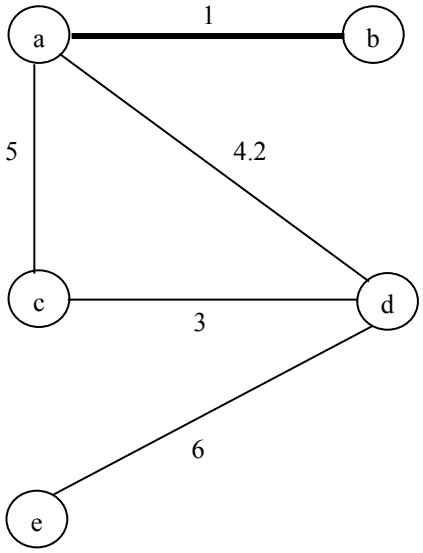


Figure: 2.6.2

Second Iteration

Next the edge (c,d) is of weight 3, minimum for the remaining edges. Also edges (a,b) and (c,d) do not form a cycle, as shown below. Therefore, (c,d) is selected so that,

$E_g = ((a,b), (c,d))$

Thus, after second iteration, the graph with selected edges in bold is as shown below:

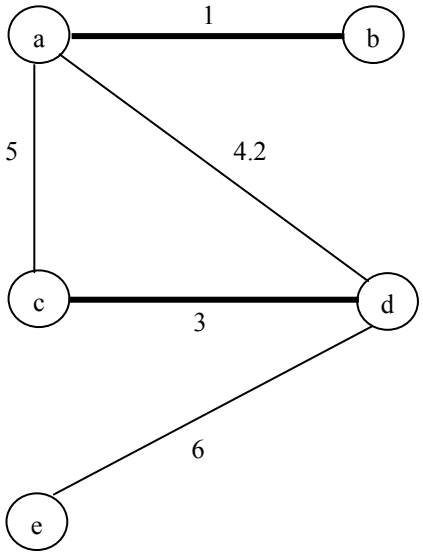


Figure: 2.6.3

It may be observed that the selected edges do not form a connected subgraph or subtree of the given graph.

Third Iteration

Next, the edge (a,d) is of weight 4.2, the minimum for the remaining edges. Also the edges in E_g alongwith the edge (a,d) do not form a cycle. Therefore, (a,d) is selected so that new $E_g = ((a,b), (c,d), (a,d))$. Thus after third iteration, the graph with selected edges in bold is as shown below.

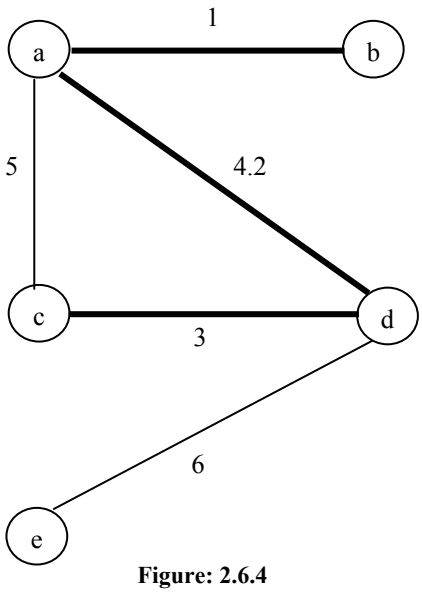


Figure: 2.6.4

Fourth Iteration

Next, the edge (a,c) is of weight 5, the minimum for the remaining edge. However, the edge (a,c) forms a cycles with two edges in E_g, viz., (a,d) and (c,d). Hence (a,c) is not selected and hence not considered as a part of the to-be-found spanning tree.

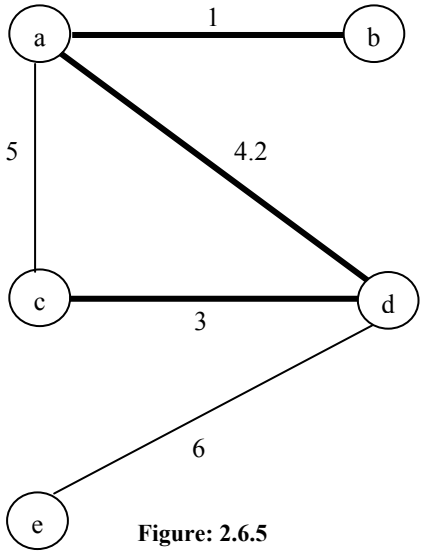


Figure: 2.6.5

At the end of fourth iteration, the graph with selected edges in bold remains the same as at the end of the third iteration, as shown below:

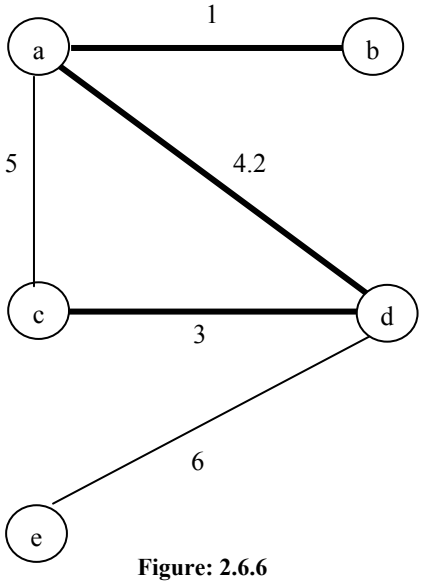


Figure: 2.6.6

Fifth Iteration

Next, the edge (e,d), the only remaining edge that can be considered, is considered. As (e,d) does not form a cycle with any of the edges in E_g. Hence the edge (e,d) is put in E_g. The graph at this stage, with selected edge in bold is as follows.

Error!

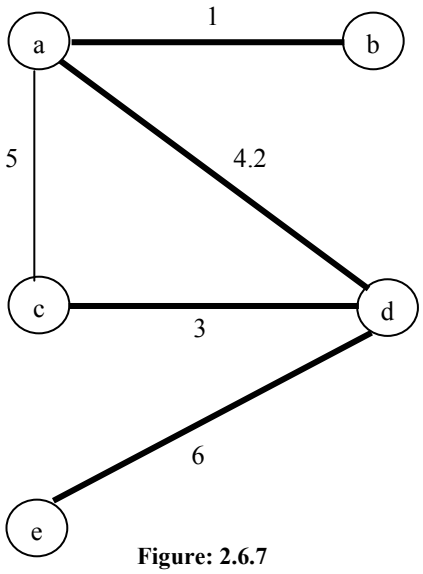


Figure: 2.6.7

At this stage we find each of the vertices of the given graph is a vertex of some edge in E_g. Further we observe that the edges in E_g form a tree, and hence, form the required spanning tree. Also, from the choice of the edges in E_g, it is clear that the spanning tree is of minimum weight. Next, we consider semi-formal definition of Kruskal's algorithm.

ALGORITHM Spanning-Kruskal (*G*)

// The algorithm constructs a minimum spanning tree by choosing successively edges
// of minimum weights out of the remaining edges.
// The input to the algorithm is a connected graph G = (V, E), in which V is the set of
// vertices and E, the set of edges and weight of each edge is also given.
// The output is the set of edges, denoted by E_T, which constitutes a minimum
// spanning tree of *G*
// the variable *edge-counter* is used to count the number of selected edges so far.
// variable t is used to count the number of edges considered so far.

Arrange the edges in *E* in nondecreasing order of the weights *of edges.* After the arrangement, the edges in order are labeled as $e_1, e_2, \ldots e_{|E|}$
$E_T \leftarrow \phi$ // initialize the set of tree edges as empty
edge-counter $\leftarrow 0$ // initialize the ecounter to zero
$t \leftarrow 0$ // initialize the number of processed edges as zero
// let n = number of edges in V

While edge-*counter* $< n-1$
    $t \leftarrow t + 1$ // increment the counter for number of edges considered so far
If *the edges* e_t *does not form a cycle with any subset of edges in* E_T **then**
    begin
    // if, e_t alongwith edges earlier in E_T do not form a cycle
    // then add e_t to E_T and increase edge counter
    $E_T \leftarrow E_T \cup \{ e_t \}$;
    edge-counter $\leftarrow$ edge-counter + 1
end if
return E_T

Summary of Kruskal's Algorithm

- The algorithm is always successful in finding a minimal spanning tree
- (Sub) tree structure may **not** be maintained, however, finally, we get a minimal spanning tree

Computing Complexity of Kruskals's

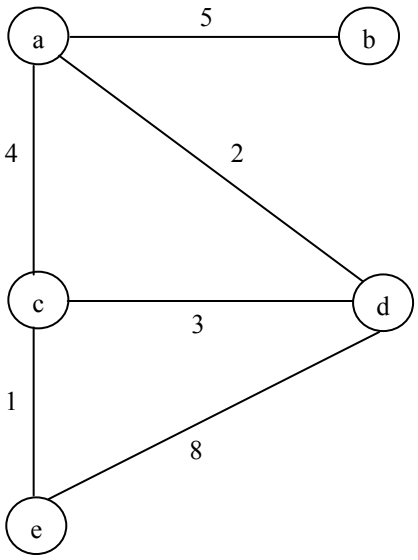
Let **a** be the number of edges and **n** be the number of nodes, initially given.
Then

(i) θ **(a log a)** time is required for sorting the edges in increasing orders of lengths

(ii) An efficient Union-Find operation takes (**2a**) find operations and (**n –1**) merge operations.

Thus Complexity of Kruskal's algorithm is O(**a log a**)

Ex. 4) Using Kruskal's algorithm, find a minimal spanning tree for the following g graph



2.7 DIJKSTRA'S ALGORITHM

Directed Graph:

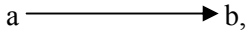
So far we have discussed applications of Greedy technique to solve problems involving undirected graphs in which each edge (a, b) from a to b is also equally an edge from b to a. In other words, the two representations (a,b) and (b,a) are for the same edge. Undirected graphs represent symmetrical relations. For example, the relation of 'brother' between male members of, say a city, is symmetric. However, in the same set, the relation of 'father' is not symmetric. Thus a general relation may be symmetric or asymmetric. A general relation is represented by a **directed** graph, in which the (directed) edge, also called an arc, (a, b) denotes an edge from a to b. However, the directed edge (a,b) is not the same as the directed edge (b,a). In the context of directed graphs, (b,a) denotes the edge from b to a. Next, we formally define a directed graph and then solve some problems, using Greedy technique, involving directed graphs.

Actually, the notation (a,b) in mathematics is used for ordered pair of the two elements viz., a and b in which a comes first and then b follows. And the ordered pair (b,a) denotes a different ordered set in which b comes first and then a follows.

However, we have misused the notation in the sense that we used the notation (a,b) to denote an unordered set of two elements, i.e., a set in which order of occurrence of a and b does not matter. In Mathematics the usual notation for an unordered set is {a,b}. In this section, we use parentheses (i.e., (and)) to denote ordered sets and braces (i.e., {and}) to denote a geneal (i.e., unordered set).

Definition:

A **directed graph or digraph** G = (V(G), E(G)) where V(G) denotes the set of vertices of G and E(G) the set of directed edges, also called arcs, of G. An arc from a to b is denoted as (a, b). Graphically it is denoted as follows:



in which the arrow indicates the direction. In the above case, the vertex a is sometimes called the **tail** and the vertex b is called the head of the arc or directed edge.

Definition:

A **Weighted Directed Graph** is a directed graph in which each arc has an assigned weight. A weighted directed graph may be denoted as G = (V(G), E(G)), where any element of E(G) may be of the form (a,b,w) where w denotes the weight of the arc (a,b). The directed Graph G = ((a, b, c, d, e), ((b, a, 3), (b, d, 2) (a, d,7), (c, b, 4), (c, d, 5), (d, e, 4), (e, c, 6))) is diagrammatically represented as follows:

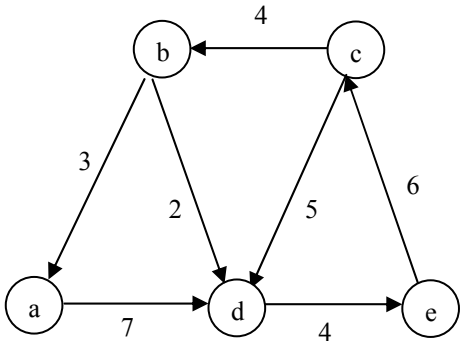


Figure: 2.7.1

Single-Source Shortest Path

Next, we consider the problem of finding the shortest distances of each of the vertices of a given weighted connected graph from some *fixed vertex* of the given graph. All the weights between pairs of vertices are taken as only positive number. The fixed vertex is called the source. The problem is known as **Single-Source Shortest Path Problem (SSSPP)**. One of the well-known algorithms for SSSPP is due to Dijkstra. The algorithm proceeds iteratively, by first consider the vertex nearest to the source. Then the algorithm considers the next nearest vertex to the source and so on. Except for the first vertex and the source, the distances of all vertices are iteratively adjusted, taking into consideration the new minimum distances of the vertices considered earlier. If a vertex is not connected to the source by an edge, then it is considered to have distance ∞ from the source.

Algorithm Single-source-Dijkstra (V,E,s)
// The inputs to the algorithm consist of the set of vertices V, the set of edges E, and s
// the selected vertex, which is to serve as the source. Further, weights w(i,j) between
// every pair of vertices i and j are given. The algorithm finds and returns d_v, the
// minimum distance of each of the vertex v in V from s. An array D of the size of
// number of vertices in the graph is used to store distances of the various vertices
// from the source. Initially Distance of the source from itself is taken as 0

// and Distance D(v) of any other vertex v is taken as ∞.
// Iteratively distances of other vertices are modified taking into consideration the
// minimum distances of the various nodes from the node with most recently modified
// distance

D(s) ← 0
For each vertex v ≠ s do
D (v) ← ∞
// Let Set-Remaining-Nodes be the set of all those nodes for which the final minimum
// distance is yet to be determined. Initially
Set-Remaining-Nodes ← V
while (Set-Remaining-Nodes ≠ ϕ) **do**
begin
choose v ∈ Set-Remaining-Nodes such that D(v) is minimum
Set-Remaining-Nodes ← Set-Remaining-Nodes ~ {v}
For each node x ∈ Set-Remaining-Nodes such that w(v, x) ≠ ∞ do
D(x) ← min {D(x), D(v) + w (v, x)}
end

Next, we consider an example to illustrate the **Dijkstra's Algorithm**

Example 2.7.1

For the purpose, let us take the following graph in which, we take a as the source

Error!

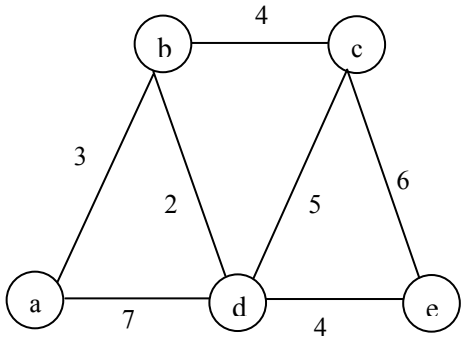
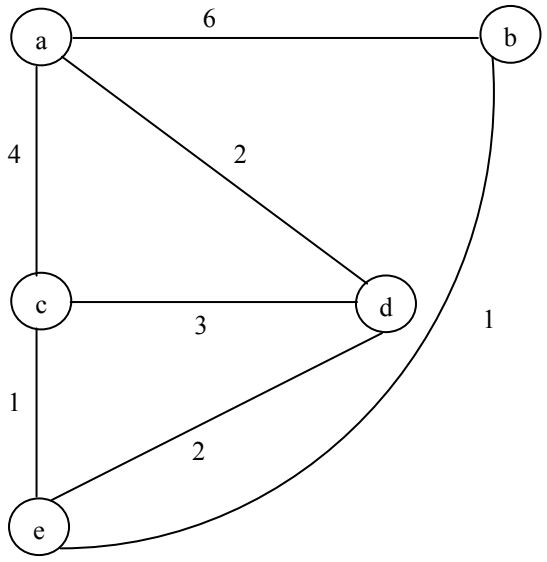


Figure: 2.7.2

Step	***Additional node***	***S = Set-of-Remaining Nodes***	***Distances from source of b, c, d, e***
Initialization	a	(b, c, d, e)	[3, ∞, 7 ∞]
1	b	(c, d, e)	[3, 3 + 4, 3 + 2, ∞]
2	d	(c, e)	[3, 3+4, 3 + 2, 3+2+4]
3	c	(e)	[3, 7, 5, 9]

For minimum distance from a, the node b is directly accessed; the node c is accessed through b; the node d is accessed through b; and the node e is accessed through b and d.

Ex. 5) Using Dijkstra's algorithm, find the minimum distances of all the nodes from node b which is taken as the source node, for the following graph.



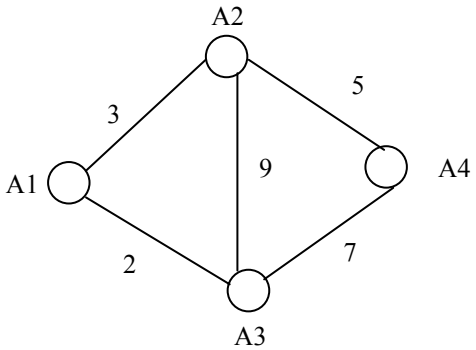
2.8 SUMMARY

In this unit, we have discussed the greedy technique **the essence of which is : In the process of solving an optimization problem, initially and at subsequent stages, evaluate the costs/benefits of the various available alternatives for the next step. Choose the alternative which is *optimal* in the sense that either it is the least costly or it is the maximum profit yielding. In this context, it may be noted that the *overall solution*, yielded by choosing *locally optimal steps*, may *not* be optimal.** Next, well-known algorithms viz., Prim's and Kruskal's that use greedy technique, to find spanning trees for connected graphs are discussed. Also Dijkstra's algorithm for solving Single-Source-Shortest path problem, again using greedy algorithm, is discussed.

2.9 SOLUTIONS/ANSWERS

Ex.1)

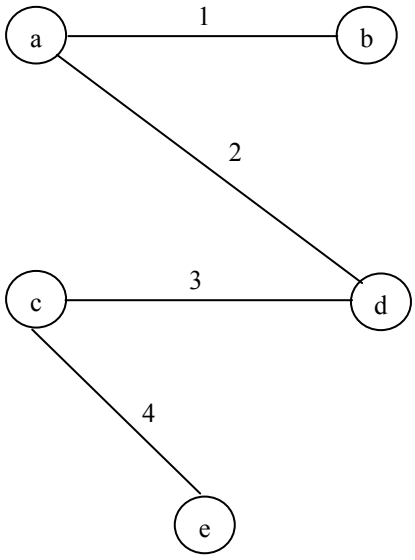
Consider the following graph, in which vertices/nodes represent cities of a country and each edge denotes a road between the cities denoted by the vertices of the edge. The label on each edge denotes the distance in 1000 kilometers between the relevant cities. The problem is to find an optimal path from A1 to A4.



Then greedy techniques suggests the route A1, A3, A4 of length 9000 kilometers, whereas the optimal path A1, A2, A4 is of length 8000 kilometers only.

Ex.2)

We will learn some systematic methods of finding a minimal spanning tree of a graph in later sections. However, by hit and trial, we get the following minimal spanning tree of the given graph.

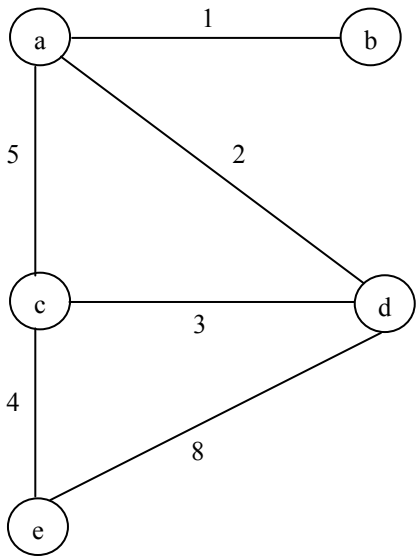


Ex.3)

The student should include the explanation on the lines of Example 2.5.1. However, the steps and stages in the process of solving the problem are as follows.

Initially

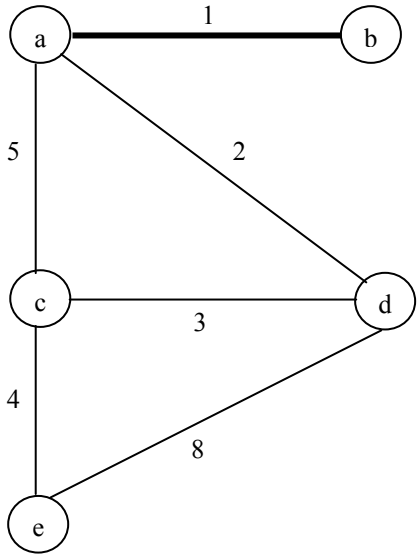
$V_T = (a)$
$E_T = \phi$



In the following figures, the edges in bold denote the chosen edges.

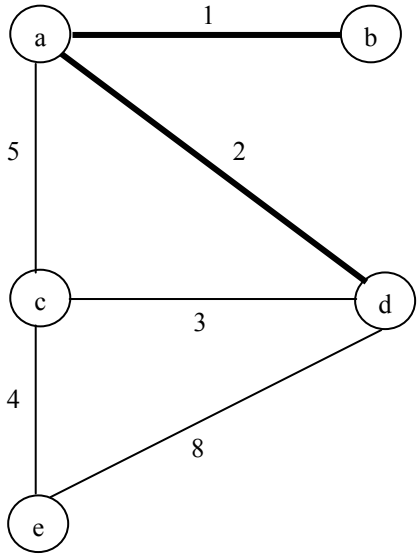
After First Iteration

V_T = (a, b)
E_T = ((a, b))



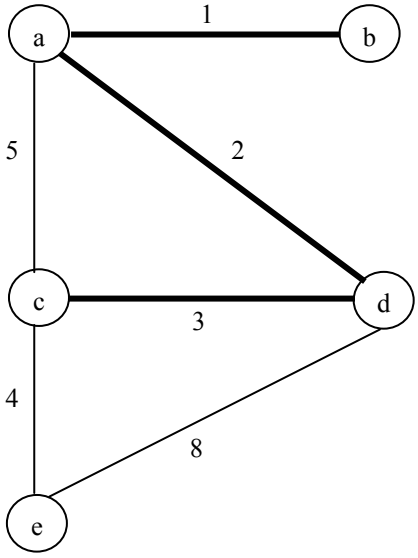
After Second Iteration

V_T = (a, b, d)
E_T = ((a, b), (a, d))



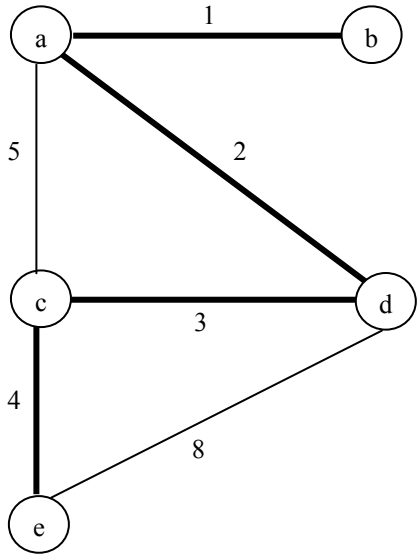
After Third Iteration

V_T = (a, b, c, d)
E_T = ((a, b), (a, d), (c, d))



After Fourth Iteration

V_T = (a, b, c, d, e)
E_T = ((a, b), (a, d), (c, d), (c, e))



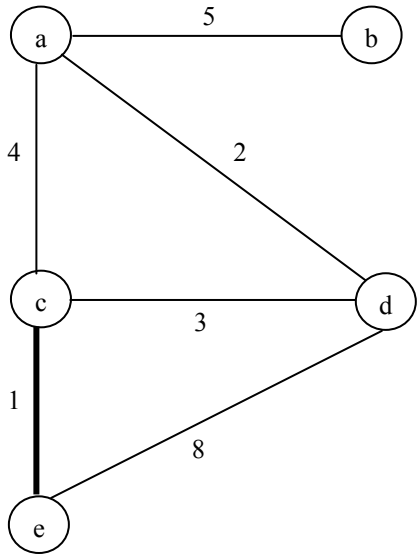
Ex. 4)

The student should include the explanation on the lines of Example 2.6.1. However, the steps and stages in the process of solving the problem are as follows:

The edges in bold denote the selected edges.

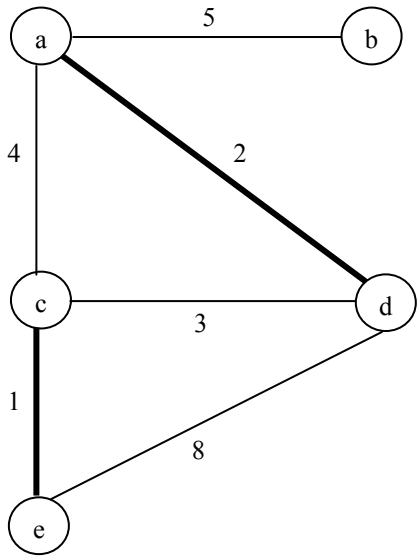
After First Iteration

E_g = ((c, e))



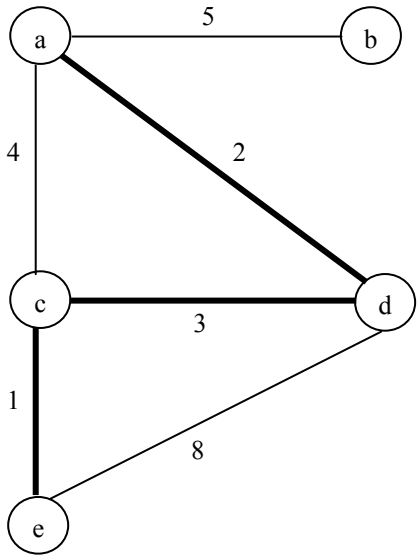
After Second Iteration

E_g = ((c, e), (a, d))



After Third Iteration

E_g = ((c, e), (a, d), (c, d))

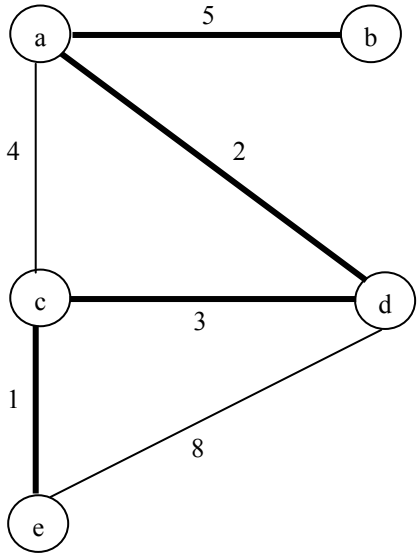


After Fourth Iteration

We can not take edge ac, because it forms a cycle with (a, d) and (c, d)

After Fifth Iteration

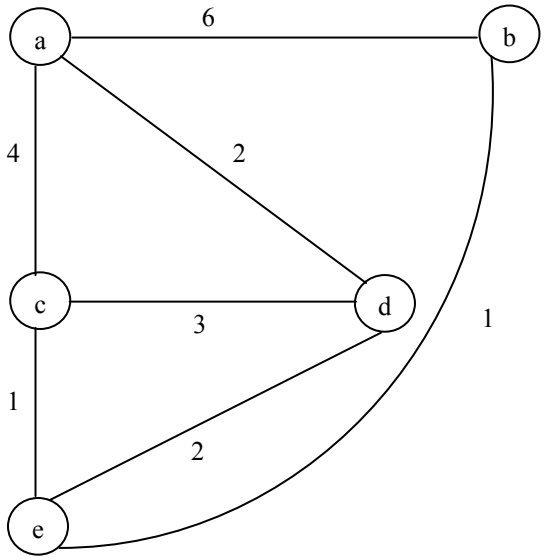
Eg = ((c, e), (a, d), (c, d), (a, b))



Now, on the above four edges all the vertices of the graph lie and these edges form a tree which is the required minimal spanning tree.

Ex. 5)

A copy of the graph is given below



Step	*Additional node*	*S = Set-of-Remaining Nodes*	*Distances from source of a, c, d, e*
Initialization	B	(a, c, d, e)	[6, ∞, ∞, 1]
1	E	(a, c, d)	[6, 2, 3 1,]
2	c	(a, d)	[6, 2, 3 1,]
3	d	(a)	[5, 2, 3, 1]

For minimum distance from b, node a is accessed through d and e; node c is accessed through e; node d is accessed through e and node e is accessed directly.

2.10 FURTHER READINGS

1. *Foundations of Algorithms,* R. Neapolitan & K. Naimipour, (D.C. Health & Company, 1996).

2. *Algoritmics: The Spirit of Computing,* D. Harel, (Addison-Wesley Publishing Company, 1987).

3. *Fundamental Algorithms* (*Second Edition*), D.E. Knuth, (Narosa Publishing House).

4. *Fundamentals of Algorithmics,* G. Brassard & P. Brately (Prentice-Hall International, 1996).

5. *Fundamentals of Computer Algorithms*, E. Horowitz & S. Sahni, (Galgotia Publications).

6. *The Design and Analysis of Algorithms,* Anany Levitin, (Pearson Education, 2003).

7. *Programming Languages* (*Second Edition*) – *Concepts and Constructs,* Ravi Sethi, (Pearson Education, Asia, 1996).