

UNIT

3

Greedy Method

GENERAL METHOD

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm*.

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm*.

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{ solution := ; // initialize the solution to empty for
  i:=1 to n do
    { x := select (a);
      if feasible (solution, x) then solution
        := Union (Solution, x);
    }
  return solution; }
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from 'a', removes it and assigns its value to 'x'. Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector. The function Union combines 'x' with solution and updates the objective function.

KNAPSACK PROBLEM

Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction x_i , $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'. The problem is stated as:

$$\begin{aligned} & \text{maximize} \quad \sum_{i=1}^n p_i x_i \\ & \text{subject to} \quad \sum_{i=1}^n w_i x_i \leq M \quad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n \end{aligned}$$

1

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of $p[i] / w[i]$ then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that  $p[i] / w[i] > p[i + 1] / w[i + 1]$ .
// m is the knapsack size and x[1: n] is the solution vector.

{ for i := 1 to n do x[i] := 0.0           // initialize x
  m;
    for i := 1 to n do
      { if (w[i] > U) then break; x[i]
        := 1.0; U := U - w[i];
      }
      if (i ≤ n) then x[i] := U / w[i];
    }
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only $O(n)$ time.

Example:

Consider the following instance of the knapsack problem: $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

1. First, we try to fill the knapsack by selecting the objects in some order:

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
1/2	1/3	1/4	$18 \times 1/2 + 15 \times 1/3 + 10 \times 1/4 = 16.5$	$25 \times 1/2 + 24 \times 1/3 + 15 \times 1/4 = 24.25$

2. Select the object with the maximum profit first ($p = 25$). So, $x_1 = 1$ and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit ($p = 24$). So, $x_2 = 2/15$

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
1	2/15	0	$18 \times 1 + 15 \times 2/15 = 20$	$25 \times 1 + 24 \times 2/15 = 28.2$

3. Considering the objects in the order of non-decreasing weights w_i .

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
0	2/3	1	$15 \times 2/3 + 10 \times 1 = 20$	$24 \times 2/3 + 15 \times 1 = 31$

4. Considered the objects in the order of the ratio p_i / w_i .

p_1/w_1	p_2/w_2	p_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / w_i . Select the object with the maximum p_i / w_i ratio, so, $x_2 = 1$ and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / w_i ratio, so $x_3 = 1/2$ and the profit earned is 7.5.

x_1	x_2	x_3	$w_i x_i$	$p_i x_i$
0	1	1/2	$15 \times 1 + 10 \times 1/2 = 20$	$24 \times 1 + 15 \times 1/2 = 31.5$

This solution is the optimal solution.

JOB SEQUENCING WITH DEADLINES

When we are given a set of 'n' jobs. Associated with each Job i , deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit p_i is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array $d[1 : n]$ is used to store the deadlines of the order of their p-values. The set of jobs $j[1 : k]$ such that $j[r]$, $1 \leq r \leq k$ are the jobs in 'j' and $d[j[1]] \leq d[j[2]] \leq \dots \leq d[j[k]]$. To test whether $J \cup$

$\{i\}$ is feasible, we have just to insert i into J preserving the deadline ordering and then verify that $d[J[r]] \leq r$, $1 \leq r \leq k+1$.

Example:

Let $n = 4$, $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$ and $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.
The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	OPTIMAL
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

Algorithm GreedyJob (d, J, n)

```
// J is a set of jobs that can be completed by their deadlines.

{
    J := {1}; for i := 2 to n do
    { if (all jobs in J U {i} can be completed by their dead lines)
      then J := J U {i};
    }
}
```

OPTIMAL MERGE PATTERNS

Given 'n' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge 'n' sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly $n + m$ record moves, the obvious choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{ treenode *
lchild; treenode * rchild;
};
```

Algorithm TREE (n)

```
// list is a global of n single node binary trees
{ for i := 1 to n - 1 do
  { pt new treenode
    (pt lchild) least (list); // merge two trees with smallest lengths
    (pt rchild) least (list);
    (pt weight) ((pt lchild) weight) + ((pt rchild) weight); insert
    (list, pt);
  }
}

return least (list); // The tree left in list is the merge tree
}
```

Example 1:

Suppose we are having three sorted files X_1 , X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

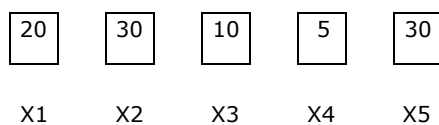
S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	$X_1 \& X_2 = T_1$	50	$T_1 \& X_3$	60	$50 + 60 = 110$
2.	$X_2 \& X_3 = T_1$	30	$T_1 \& X_1$	60	$30 + 60 = 90$

The Second case is optimal.

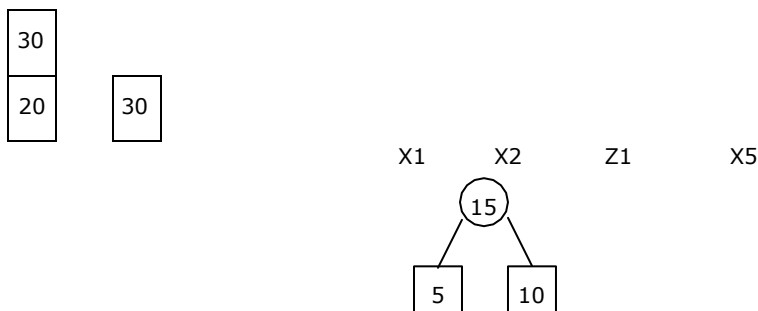
Example 2:

Given five files (X_1, X_2, X_3, X_4, X_5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.

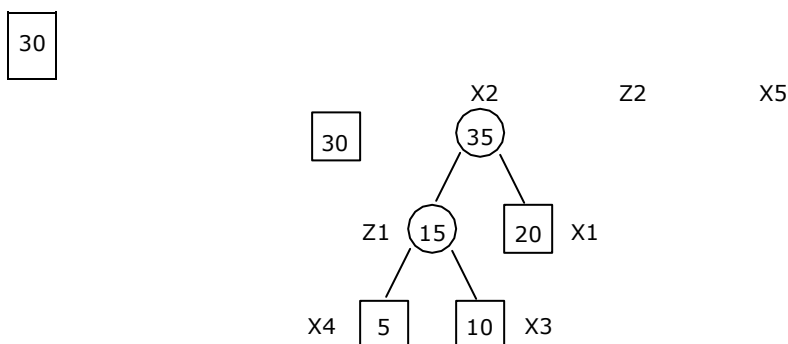
Solution:



Merge X_4 and X_3 to get 15 record moves. Call this Z_1 .



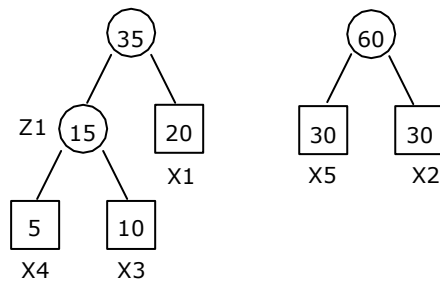
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2 .



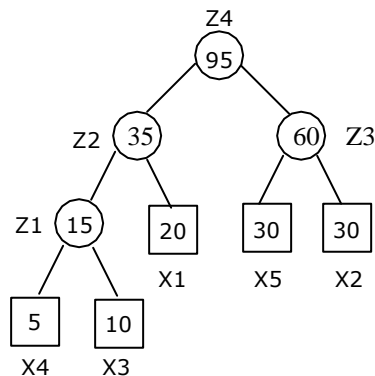
Merge X_2 and X_5 to get 60 record moves. Call this Z_3 .

Z_2

Z_3



Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4 .



Therefore the total number of record moves is $15 + 35 + 60 + 95 = 205$. This is an optimal merge pattern for the given problem.

Huffman Codes

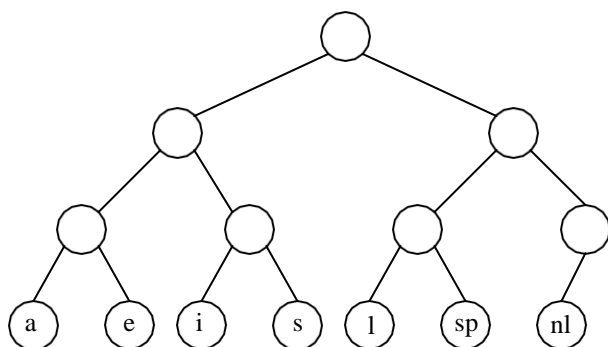
Another application of Greedy Algorithm is file compression.

Suppose that we have a file only with characters a, e, i, s, t, spaces and new lines, the frequency of appearance of a's is 10, e's fifteen, twelve i's, three s's, four t's, thirteen banks and one newline.

Using a standard coding scheme, for 58 characters using 3 bits for each character, the file requires 174 bits to represent. This is shown in table below.

<u>Character</u>	<u>Code</u>	<u>Frequency</u>	<u>Total bits</u>
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
Space	101	13	39
New line	110	1	3

Representing by a binary tree, the binary code for the alphabets are as follows:



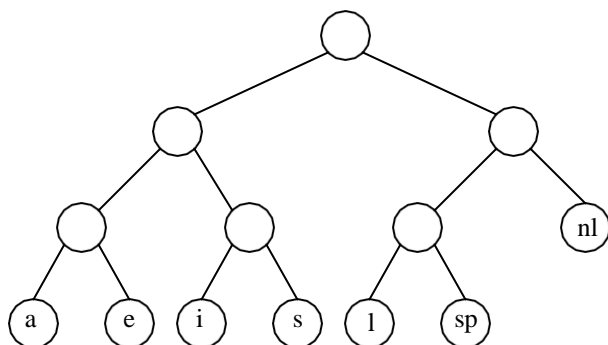
The representation of each character can be found by starting at the root and recording the path. Use a 0 to indicate the left branch and a 1 to indicate the right branch.

If the character c_i is at depth d_i and occurs f_i times, the cost of the code is equal to

$$d_i f_i$$

With this representation the total number of bits is $3 \times 10 + 3 \times 15 + 3 \times 12 + 3 \times 3 + 3 \times 4 + 3 \times 13 + 3 \times 1 = 174$

A better code can be obtained by with the following representation.



The basic problem is to find the full binary tree of minimal total cost. This can be done by using Huffman coding (1952).

Huffman's Algorithm:

Huffman's algorithm can be described as follows: We maintain a forest of trees. The weights of a tree is equal to the sum of the frequencies of its leaves. If the number of characters is 'c'. $c - 1$ times, select the two trees T_1 and T_2 , of smallest weight, and form a new tree with sub-trees T_1 and T_2 . Repeating the process we will get an optimal Huffman coding tree.

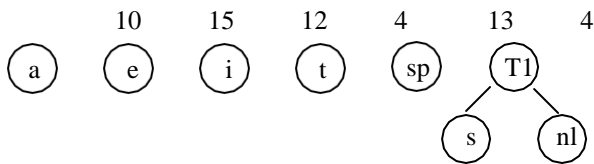
Example:

The initial forest with the weight of each tree is as follows:

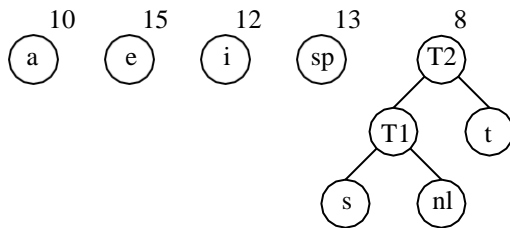
10	15	12	3	4	13	1	a
e	i	s	t	sp	nl		

The two trees with the lowest weight are merged

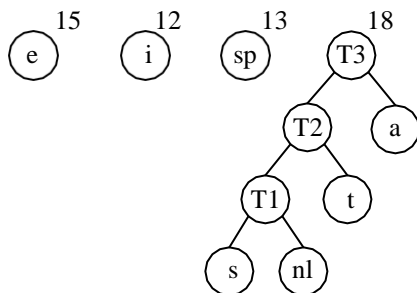
together, creating the forest, the Huffman algorithm after the first merge with new root T_1 is as follows: The total weight of the new tree is the sum of the weights of the old trees.



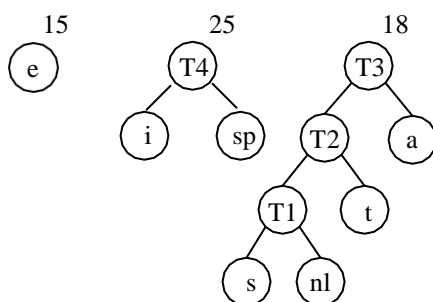
We again select the two trees of smallest weight. This happens to be T_1 and t , which are merged into a new tree with root T_2 and weight 8.



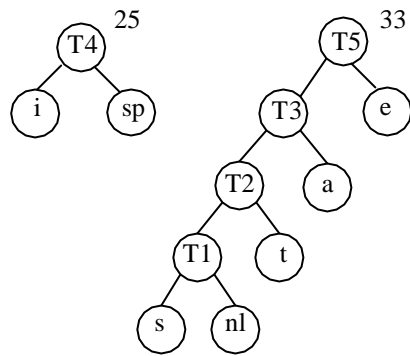
In next step we merge T_2 and a creating T_3 , with weight $10+8=18$. The result of this operation is



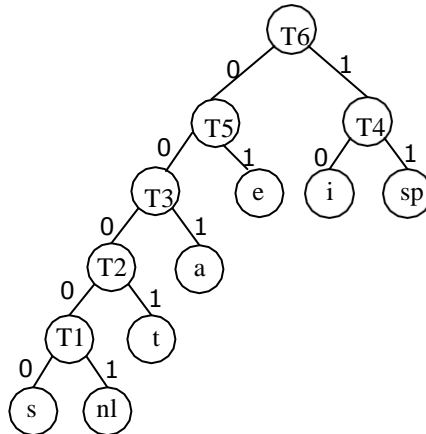
After third merge, the two trees of lowest weight are the single node trees representing i and the blank space. These trees merged into the new tree with root T_4 .



The fifth step is to merge the trees with roots e and T_3 . The results of this step is



Finally, the optimal tree is obtained by merging the two remaining trees. The optimal trees with root T_6 is:



The full binary tree of minimal total cost, where all characters are obtained in the leaves, uses only 146 bits.

Character	Code	Frequency	Total bits (Code bits X frequency)
A	001	10	30
E	01	15	30
I	10	12	24
S	00000	3	15
T	0001	4	16
Space	11	13	26
New line	00001	1	5
		Total :	146

GRAPH ALGORITHMS

Basic Definitions:

Graph G is a pair (V, E) , where V is a finite set (set of vertices) and E is a finite

set of pairs from V (set of edges). We will often denote $n := |V|$, $m := |E|$.

Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If $(u, v) \in E$, then vertices u , and v are adjacent.

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called **weighted**.

Degree of a vertex v is the number of vertices u for which $(u, v) \in E$ (denote $\deg(v)$). The number of **incoming edges** to a vertex v is called **in-degree** of the vertex (denote $\text{indeg}(v)$). The number of **outgoing edges** from a vertex is called **out-degree** (denote $\text{outdeg}(v)$).

Representation of Graphs:

Consider graph $G = (V, E)$, where $V = \{v_1, v_2, \dots, v_n\}$.

Adjacency matrix represents the graph as an $n \times n$ matrix $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E, \\ 0, & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

$$a_{i,j} = \begin{cases} w(v_i, v_j), & \text{if } (v_i, v_j) \in E, \\ \text{default}, & \text{otherwise,} \end{cases}$$

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be , meaning value larger than any other value).

Adjacency List: An array $\text{Adj}[1 \dots n]$ of pointers where for $1 \leq v \leq n$, $\text{Adj}[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

	1	2	3
1	1	1	1
2	0	0	1
3	0	1	0

1

→

1

→

2

→

3

↘

2

→

3

↘

3

→

2

↘

Adjacency matrix

Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, \dots, v_k) , where for all i , $(v_i, v_{i+1}) \in E$. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices $(v_1, v_2, \dots, v_k, v_{k+1} = v_1)$, where for all i , $(v_i, v_{i+1}) \in E$ and all vertices in the cycle are distinct except pair v_1, v_{k+1} .

Subgraphs and Spanning Trees:

Subgraphs: A graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v . If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph $G = (V, E)$ is a tree that contains all vertices of V and is a subgraph of G . A single graph can have multiple spanning trees.

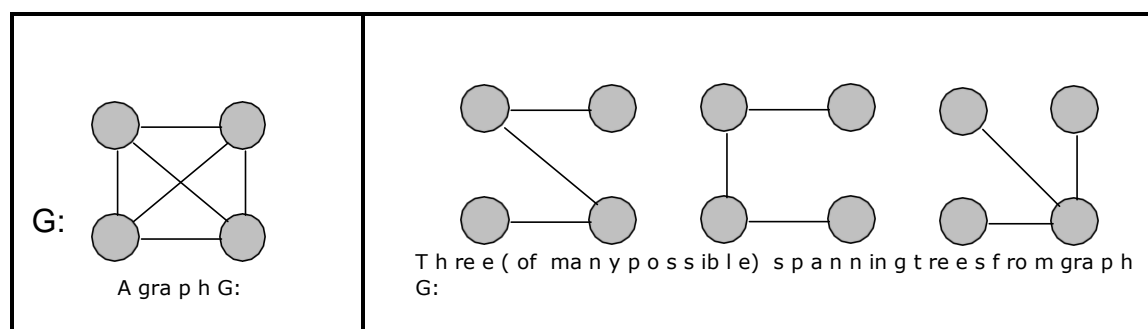
Lemma 1: Let T be a spanning tree of a graph G . Then

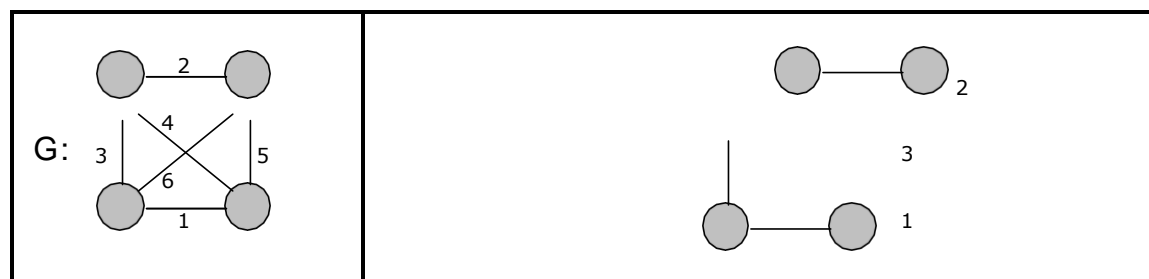
1. Any two vertices in T are connected by a unique simple path.
2. If any edge is removed from T , then T becomes disconnected.
3. If we add any edge into T , then the new graph will contain a cycle.
4. Number of edges in T is $n-1$.

Minimum Spanning Trees (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree $w(T)$ is the sum of weights of all edges in T . The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.





A weighted graph G:

The minimal spanning tree from weighted graph G:

Here are some examples:

To explain further upon the Minimum Spanning Tree, and what it applies to, let's consider a couple of real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until $(n - 1)$ edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

Algorithm Kruskal (E, cost, n, t)

// E is the set of edges in G. G has n vertices. cost [u, v] is the
 // cost of edge (u, v). 't' is the set of edges in the minimum-cost
 spanning tree. // The final cost is returned.

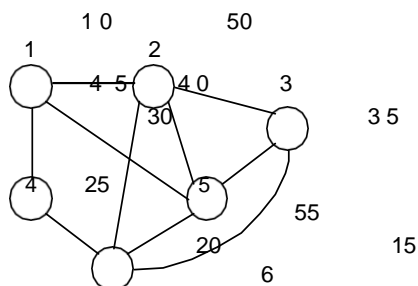
```
{
    Construct a heap out of the edge costs using
    heapify; for i := 1 to n do parent [i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from
        the heap and re-heapify using Adjust; j := Find (u); k
        := Find (v);
        if (j ≠ k) then
        { i := i + 1;
          t [i, 1] := u; t [i, 2] := v;
          mincost := mincost + cost [u, v];
          Union (j, k);
        }
    }
    if (i = n-1) then write ("no spanning
    tree"); else return mincost;
}
```

Running time:

The number of finds is at most $2e$, and the number of unions at most $n-1$. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than $O(n + e)$.

We can add at most $n-1$ edges to tree T . So, the total time for operations on T is $O(n)$.

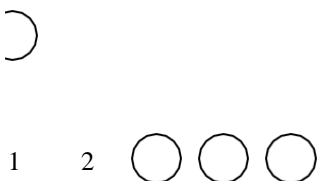

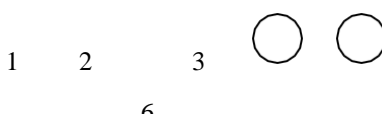
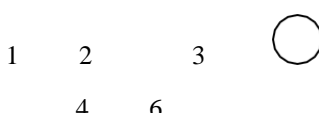
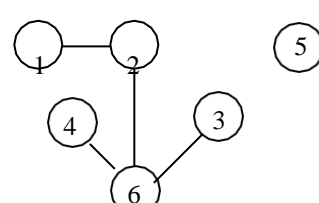
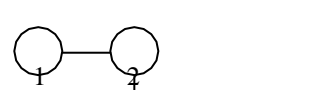
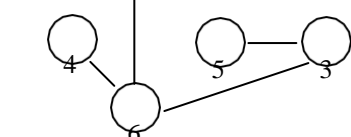
Summing up the various components of the computing times, we get $O(n + e \log e)$ as asymptotic complexity

Example 1:

Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Finds** on the vertex sets suffice. When an edge is included in T , two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$	
(1, 2)	10		$\{1, 2\}, \{3\}, \{4\}, \{5\}, \{6\}$	The vertices 1 and 2 are in different sets, so the edge is combined
(3, 6)	15		$\{1, 2\}, \{3, 6\}, \{4\}, \{5\}$	The vertices 3 and 6 are in different sets, so the edge is combined
(4, 6)	20		$\{1, 2\}, \{3, 4, 6\}, \{5\}$	The vertices 4 and 6 are in different sets, so the edge is combined
(2, 6)	25		$\{1, 2, 3, 4, 6\}, \{5\}$	The vertices 2 and 6 are in different sets, so the edge is combined
(1, 4)	30	 Reject		The vertices 1 and 4 are in the same set, so the edge is rejected
(3, 5)	35		$\{1, 2, 3, 4, 5, 6\}$	The vertices 3 and 5 are in the same set, so the edge is combined

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

Algorithm Algorithm Prim

(E, cost, n, t)

```
// E is the set of edges in G. cost [1:n, 1:n] is the
cost // adjacency matrix of an n vertex graph such that cost [i,
j] is
// either a positive real number or if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is
an edge in // the minimum-cost spanning tree. The final cost is
returned.
{
  Let (k, l) be an edge of minimum cost in
  E; mincost := cost [k, l]; t [1, 1] := k; t
  [1, 2] := l;
    for i := 1 to n do // Initialize near
      if (cost [i, l] < cost [i, k]) then near
      [i] := l; else near [i] := k;
      near [k] := near [l] := 0; for i:=2 to n - 1 do //
Find n - 2 additional edges for t.
  {
    Let j be an index such that near [j] 0 and cost
    [j, near [j]] is minimum; t [i, 1] := j; t [i, 2] := near [j];
    mincost := mincost + cost [j, near [j]]; near [j] := 0 for
    k:= 1 to n do // Update near[.
      if ((near [k] 0) and (cost [k, near [k]] > cost [k, j])) then
        near [k] := j;
    }
  return mincost;
}
```

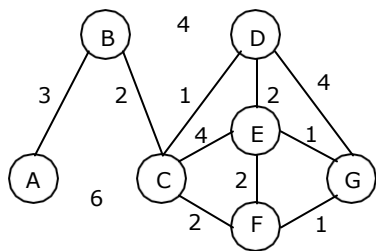

Running time:

We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get O

(n²) time when we implement dist with array, O (n + E log n) when we implement it with a heap.

EXAMPLE 1:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



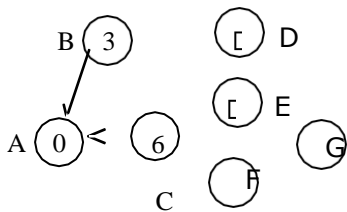
SOLUTION:

.
. .
.

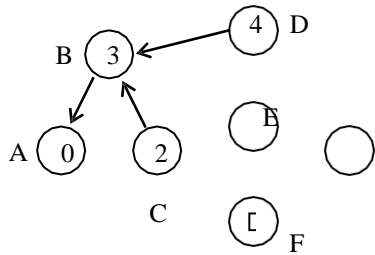
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6		?		?
Next	*	A	A	A	A	A	A

The stepwise progress of the prim's algorithm is as follows:

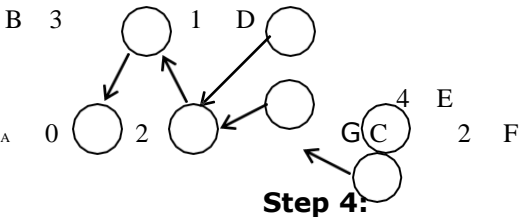
Step 1:



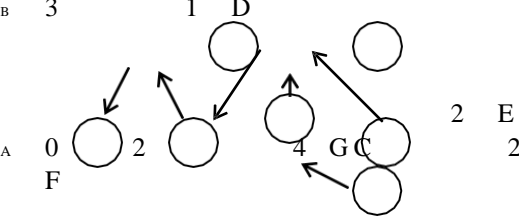
Step 2:



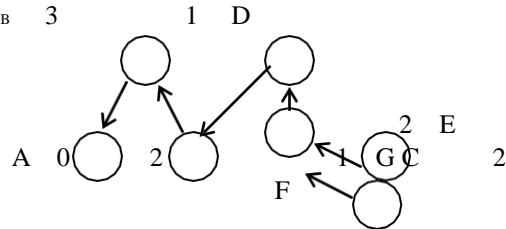
Step 3:



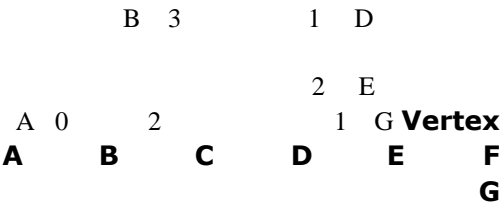
Step 4:



Step 5:



Step 6:



Step 7:



Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	?	?	?
Next	*	A	B	B	A	A	A

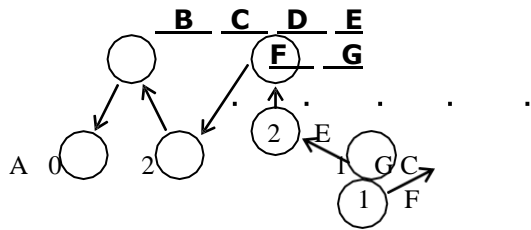
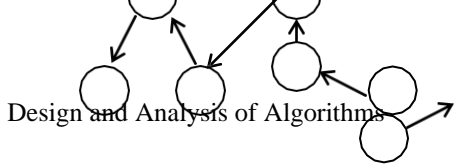
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	?
Next	*	A	B	C	C	C	A

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Vertex	A	B	C	D	E	F
G	0	0	0	0	0	1
Status	0	0	0	0	0	1
Dist.	0	3	2	1	2	1
Next	*	A	B	C	D	G

Vertex	A
Status	0
Dist.	0
Next	*

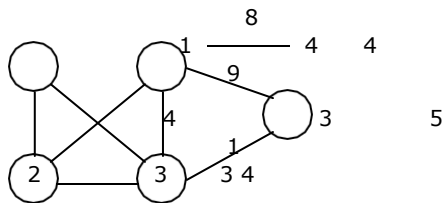


0 0 0 0
0 0 0 0

EXAMPLE 2:

A 3 B 2 C 1 D 2 G 1 E

Considering the following graph, find the minimal spanning tree using prim's algorithm.

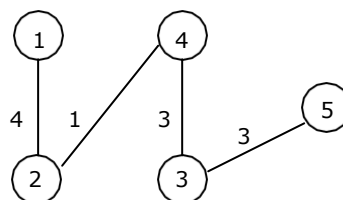


The cost adjacent matrix is

4	9	8	?	?
?	4	4	1	?
?	9	4	3	?
8	1	3	4	?
?	?	?	3	4

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2



The cost of Minimal spanning tree = 11.

The steps as per the algorithm are as follows:

Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

$K = 2, l = 4$
 $\text{Min cost} = \text{cost}(2, 4) = 1$

$T[1, 1] = 2$

$T[1, 2] = 4$

for i = 1 to 5	Near matrix	Edges added to min spanning tree:										
Begin	<table><tr><td>2</td><td></td><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2					1	2	3	4	5	$T[1, 1] = 2$ $T[1, 2] = 4$
2												
1	2	3	4	5								
i = 1 is cost (1, 4) < cost (1, 2) 8 < 4, No Than near (1) = 2	<table><tr><td>2</td><td>4</td><td></td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	4				1	2	3	4	5	
2	4											
1	2	3	4	5								
i = 2 is cost (2, 4) < cost (2, 2) 1 < , Yes So near [2] = 4	<table><tr><td>2</td><td>4</td><td>4</td><td></td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	4	4			1	2	3	4	5	
2	4	4										
1	2	3	4	5								
i = 3 is cost (3, 4) < cost (3, 2) 1 < 4, Yes So near [3] = 4	<table><tr><td>2</td><td>4</td><td>4</td><td>2</td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	4	4	2		1	2	3	4	5	
2	4	4	2									
1	2	3	4	5								
i = 4 is cost (4, 4) < cost (4, 2) < 1, no So near [4] = 2	<table><tr><td>2</td><td>4</td><td>4</td><td>2</td><td>4</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	4	4	2	4	1	2	3	4	5	
2	4	4	2	4								
1	2	3	4	5								
i = 5 is cost (5, 4) < cost (5, 2) 4 < , yes So near [5] = 4	<table><tr><td>2</td><td>0</td><td>4</td><td>0</td><td>4</td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	2	0	4	0	4	1	2	3	4	5	
2	0	4	0	4								
1	2	3	4	5								
end near [k] = near [l] = 0												

near [2] = near[4] = 0		
<p>for i = 2 to n-1 (4) do</p> <p>i = 2</p> <p>for j = 1 to 5 j = 1 near(1)0 and cost(1, near(1)) 2 0 and cost (1, 2) = 4</p> <p>i = 2 near (2) = 0</p> <p>j = 3 is near (3) 0 4 0 and cost (3, 4) = 3</p>		

<p>j = 4 near (4) = 0</p> <p>J = 5 Is near (5) 0 4 0 and cost (4, 5) = 4</p> <p>select the min cost from the above obtained costs, which is 3 and corresponding J = 3</p> <p>min cost = 1 + cost(3, 4) = 1 + 3 = 4</p> <p>T (2, 1) = 3 T (2, 2) = 4</p> <p>Near [j] = 0 i.e. near (3) =0</p> <p><u>for (k = 1 to n)</u></p> <p>K = 1 is near (1) 0, yes 2 0 and cost (1,2) > cost(1, 3) 4 > 9, No</p> <p>K = 2 Is near (2) 0, No</p> <p>K = 3 Is near (3) 0, No</p> <p>K = 4 Is near (4) 0, No</p> <p>K = 5 Is near (5) 0 4 0, yes is cost (5, 4) > cost (5, 3) 4 > 3, yes than near (5) = 3 <u>i =</u> <u>3</u></p> <p><u>for (j = 1 to 5)</u> J = 1 is near (1) 0 2 0 cost (1, 2) = 4</p> <p>J = 2 Is near (2) 0, No</p>	<table><tr><td>2</td><td>0</td><td>0</td><td>0</td><td>4</td></tr></table> <p>1 2 3 4 5</p> <table><tr><td>2</td><td>0</td><td>0</td><td>0</td><td>3</td></tr></table> <p>1 2 3 4 5</p>	2	0	0	0	4	2	0	0	0	3	<p>T (2, 1) = 3 T (2, 2) = 4</p>
2	0	0	0	4								
2	0	0	0	3								

$J = 3$
Is near (3) 0, no Near
 $(3) = 0$

$J = 4$
Is near (4) 0, no Near
 $(4) = 0$

$J = 5$
Is near (5) 0
Near (5) = 3 3 0, yes And
cost (5, 3) = 3

Choosing the min cost from the
above obtaining costs
which is 3 and corresponding J
= 5

Min cost = $4 + \text{cost}(5, 3)$
= $4 + 3 = 7$

$T(3, 1) = 5$
 $T(3, 2) = 3$

Near (J) = 0 near (5) = 0

for (k=1 to 5)

$k = 1$
is near (1) 0, yes and
 $\text{cost}(1,2) > \text{cost}(1,5)$
 $4 > , \text{No}$

$K = 2$
Is near (2) 0 no

$K = 3$
Is near (3) 0 no

$K = 4$
Is near (4) 0 no

$K = 5$
Is near (5) 0 no

i = 4

for J = 1 to 5 J
= 1

2	0	0	0	0
1	2	3	4	5

$T(3, 1) = 5$
 $T(3, 2) = 3$

<p>Is near (1) 0 2 0, yes cost (1, 2) = 4</p> <p>i = 2 is near (2) 0, No</p>		
---	--	--

J = 3
Is near (3) 0, No Near
(3) = 0

J = 4
Is near (4) 0, No Near
(4) = 0

J = 5
Is near (5) 0, No Near
(5) = 0

Choosing min cost from the
above it is only '4' and
corresponding J = 1

Min cost = 7 + cost (1,2)
 = 7+4 = 11

T (4, 1) = 1
T (4, 2) = 2

Near (J) = 0 Near (1) = 0 for
(k = 1 to 5)

K = 1
Is near (1) 0, No

K = 2
Is near (2) 0, No

K = 3
Is near (3) 0, No

K = 4
Is near (4) 0, No

K = 5
Is near (5) 0, No

End.

0	0	0	0	0
1	2	3	4	5

T (4, 1) = 1
T (4, 2) = 2

4.8.7. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

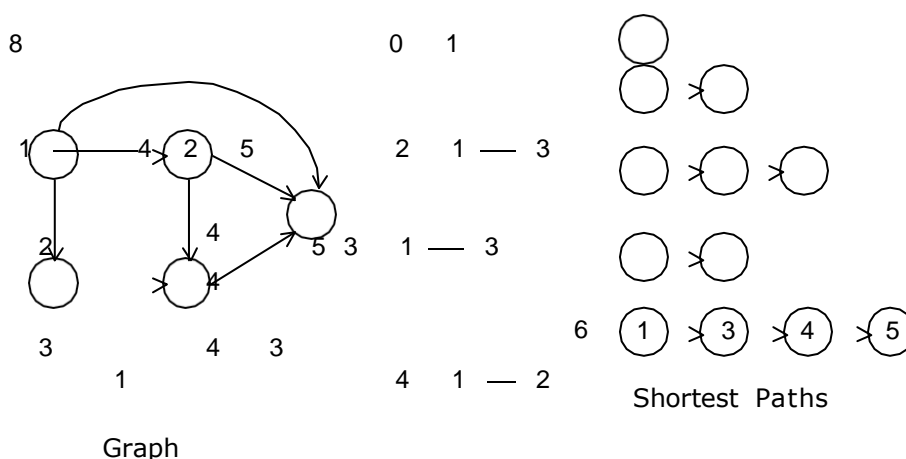
In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between them (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Algorithm:

Algorithm Shortest-Paths (v, cost, dist, n)

```
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{ for i := 1 to n do
    {
        S [i] := false; // Initialize S.
        dist [i] := cost [v, i];
    }
    S[v] := true; dist[v] := 0.0; // Put v in S.
    for num := 2 to n - 1 do
    {
        Determine n - 1 paths from v.
        Choose u from among those vertices not in S such that dist[u] is
        minimum;
        S[u] := true; // Put u in S.
        for (each w adjacent to u with S [w] = false) do
            if (dist [w] > (dist [u] + cost [u, w])) then // Update distances
                dist [w] := dist [u] + cost [u, w];
    }
}
```

Running time:

Depends on implementation of data structures for dist.

Build a structure with n elements A
 at most $m = E$ times decrease the value of an item mB

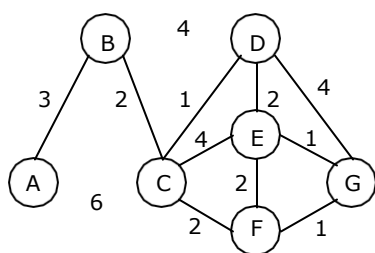
' n ' times select the smallest value nC

For array $A = O(n)$; $B = O(1)$; $C = O(n)$ which gives $O(n^2)$ total.

For heap $A = O(n)$; $B = O(\log n)$; $C = O(\log n)$ which gives $O(n + m \log n)$
 total.

Example 1:

Use Dijkstras algorithm to find the shortest path from A to each of the other six vertices in the graph:



Solution:

	0	3	6	-	-	-	-	?
?								?
	3	0	2	4	-	-	-	?
	6	2	0	1	4	2	-	?
	4	1	0	2	-	-	?	4
	-	4	2	0	2	1	?	
	-	-	2	-	2	0	1	
	-	-	-	-	4	1	1	0

Here - means infinite

The problem is solved by considering the following information:

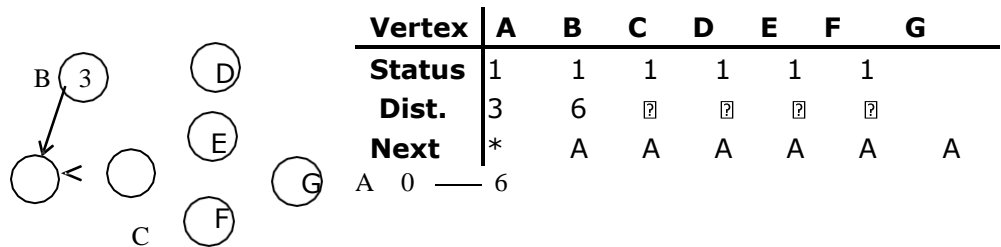
has Status[v] will be either '0', meaning that the shortest path from v to v_0
 definitely been found; or '1', meaning that it hasn't.

Dist[v] will be a number, representing the length of the shortest path from v to v_0 found so far.

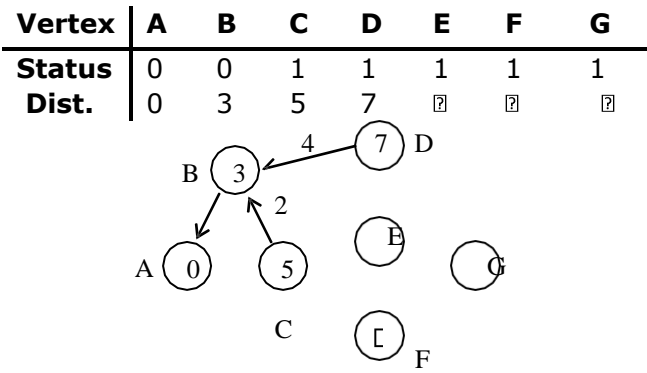
Next[v] will be the first vertex on the way to v_0 along the shortest path found so far from v to v_0

The progress of Dijkstra’s algorithm on the graph shown above is as follows:

Step 1:

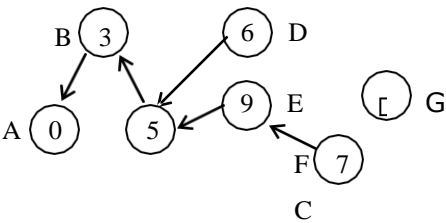


Step 2:

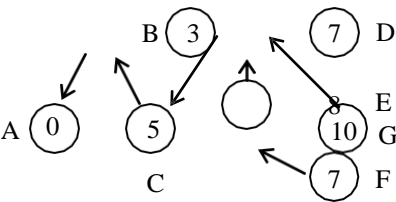


Next * A B B A A A

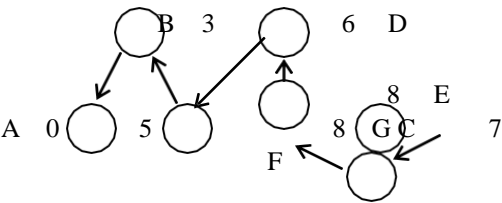
Step 3:



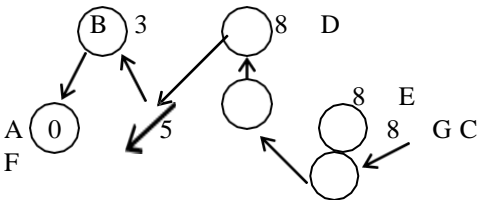
Step 4:



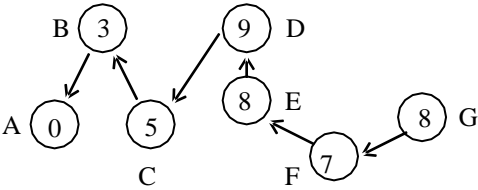
Step 5:



Step 6:



Step 7:



Vertex A		B	C	D	E	F	
		G					
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	10
Next	*	A	B	C	C	C	A

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Vertex	A	B	C	D	E	F	G
--------	---	---	---	---	---	---	---

Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Vertex	A
Status	0
Dist.	0
Next	*

	B				
C	D	E	F	G	
	0	0	0	0	0
1					
3	5	6	8	7	
8					
A	B	C	D	C	
F					

		Vertex	A	B	C	D
E	F	G				
Status	0		0	0	0	0
			0			
Dist.			0	3	5	6
8					8	7
Next			*	A	B	C
			F			