# Chapter-3
# Greedy Method

## 3.1 Greedy Technique Definition

Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are: feasible, i.e. satisfying the constraints locally optimal (with respect to some neighborhood definition) greedy (in terms of some measure), and irrevocable. For some problems, it yields a globally optimal solution for every instance. For most, does not but can be useful for fast approximations. We are mostly interested in the former case in this class.

**Generic Algorithm**

```
Algorithm Greedy(a,n)
{
//a[1..n] contains the n inputs.
solution:= ∅;
For i:= 1to n do
{
X=select(a);
If Feasible(solution , x) then
solution:= union(solution, x);
}
return solution;
}
```

**Applications of the Greedy Strategy**

Optimal solutions:

- change making for “normal” coin denominations
- minimum spanning tree (MST)
- single-source shortest paths
- simple scheduling problems
- Huffman codes

Approximations/heuristics:

- Traveling salesman problem (TSP)
- knapsack problem
- other combinatorial optimization problems

Change-Making Problem:
Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$, give change for amount $n$ with the least number of coins
Example: $d_1 = 25c$, $d_2 = 10c$, d3 = 5c, d4 = 1c and n = 48c

Greedy solution:
Greedy solution is Optimal for any amount and “normal’’ set of denominations
Ex: Prove the greedy algorithm is optimal for the above denominations. It may not be optimal for arbitrary coin denominations.

**3.2 The Fractional Knapsack Problem**

Given a set S of $n$ items, with each item i having $b_i$ - a positive benefit $w_i$ - a positive weight our goal is to Choose items with maximum total benefit but with weight at most $W$. If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**. In this case, we let $x_i$ denote the amount we take of item i

Objective: maximize

$$\sum_{i \in S} b_i (x_i / w_i)$$

Constraint:

$$\sum_{i \in S} x_i \leq W$$

**Algorithm for greedy strategy for knapsack problem:**

**Algorithm** Greedy***Knapsack***(m,n)

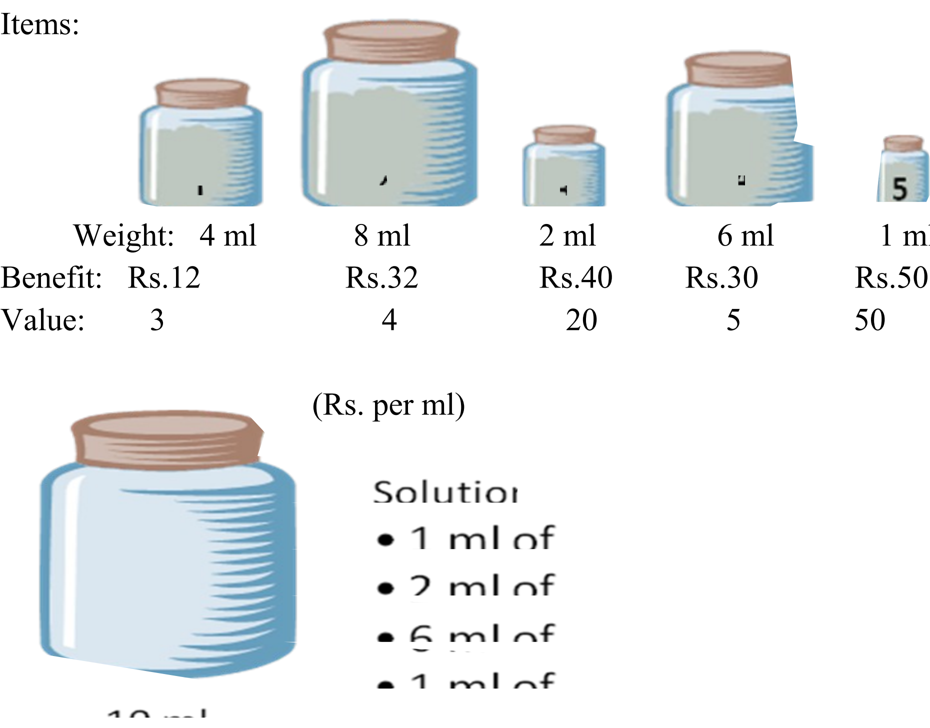
//p[1:n] and w[1:n] contain profits and weights respectively of n objects ordered such that //p[i]/w[i] >=p[i+1]/w[i+1].m is the knapsack size and x[1:n] is the solution vector
{
For i:= 1 to n do x[i]:=0.0; //initialize x
U:=m;

```
{
If (w[i]>U) then break;
x[i]:=1.0; U:=U-w[i];
}
If (i<=n) then x[i]:=U/w[i];
}
```

**Example model-1**

In this model items are arranged by their values, maximum selected first, process continuous till minimum value. Here given a set S of n items, with each item i having $b_i$ - a positive benefit $w_i$ - a positive weight here our goal is to Choose items with maximum total benefit but with weight at most W.

Items:




| | | | | | |
|---|---|---|---|---|---|
| Weight: | 4 ml | 8 ml | 2 ml | 6 ml | 1 ml |
| Benefit: | Rs.12 | Rs.32 | Rs.40 | Rs.30 | Rs.50 |
| Value: | 3 | 4 | 20 | 5 | 50 |

(Rs. per ml)

**Knapsack Problem model-2**

In this model items are arranged by their weights, lightest weight selected first, process continuous till the maximum weight. You have a knapsack that has capacity (weight) and You have several items $I_1,\ldots,I_n$..Each item $I_j$ has a weight $w_j$ and a benefit $b_j$.You want to place a certain number of copies of each item $I_j$ in the knapsack so that:

i)The knapsack weight capacity is not exceeded and

ii)The total benefit is maximal.

Example

| Item | Weight | Benefit |
|---|---|---|
| A | 2 | 60 |
| B | 3 | 75 |
| C | 4 | 90 |

f (0), f(1)
f (0) = 0. Why? The knapsack with capacity 0 can have nothing in it.
f (1) = 0. There is no item with weight 1.
f (2)
f (2) = 60. There is only one item with weight 60.then choose A.
f (3)
f(3) = MAX $\{b_j + f(w-w_j) \mid I_j$ is an item$\}$.
= MAX {60+f (3-2), 75 + f (3-3)}
= MAX {60 + 0, 75 + 0}
= 75 then Choose B**.**
F (4)
F (4) = MAX $\{b_j + f(w-w_j) \mid I_j$ is an item$\}$.
= MAX {60 + f (4-2), 75 + f (4-3), 90+f (4-4)}
= MAX {60 + 60, 75 + f (1), 90 + f (0)}
= MAX {120, 75, 90}
=120. Then choose A
F (5)
F (5) = MAX $\{b_j + f(w-w_j) \mid I_j$ is an item$\}$.
= MAX {60 + f (5-2), 75 + f (5-3), 90+f (5-4)}

= MAX {60 + f (3), 75 + f (2), 90 + f (1)}
= MAX {60 + 75, 75 + 60, 90+0}
= 135. Then choose A or B.

Result

Optimal knapsack weight is 135. There are two possible optimal solutions:
Choose A during computation of f (5).
Choose B in computation of f (3).
Choose B during computation of f (5).
Choose A in computation of f (2).
Both solutions coincide. Take A and B.

**Procedure to solve the knapsack problem**

It is Much easier for item $I_j$, let $r_j = b_j / w_j$. This gives you the benefit per measure of weight and then Sort the items in descending order of $r_j$ .Pack the knapsack by putting as many of each item as you can walking down the sorted list.

**Example model-3**

I=<I1,I2,I3,I4,I5> W=<5,10,20,30,40> V=<30,20,100,90,160> knapsack capacity W=60, the solution to the fractional knapsack problem is given as:
Initially

| Item | Wi | Vi |
|---|---|---|
| I1 | 5 | 30 |
| I2 | 10 | 20 |
| I3 | 20 | 100 |
| I4 | 30 | 90 |
| I5 | 40 | 160 |

Taking value per weight ratio

| Item | wi | vi | Pi=vi/wi |
|---|---|---|---|
| I1 | 5 | 30 | 6.0 |
| I2 | 10 | 20 | 2.0 |
| I3 | 20 | 100 | 5.0 |
| I4 | 30 | 90 | 3.0 |
| I5 | 40 | 160 | 4.0 |

Arranging item with decreasing order of Pi

| Item | wi | vi | Pi=vi/wi |
|---|---|---|---|
| I1 | 5 | 30 | 6.0 |
| I2 | 20 | 100 | 5.0 |
| I3 | 40 | 160 | 4.0 |
| I4 | 30 | 90 | 3.0 |
| I5 | 10 | 20 | 2.0 |

Filling knapsack according to decreasing value of Pi, max. value = v1+v2+new (v3)=30+100+140=270

### 3.3 Greedy Method – Job Sequencing Problem

Job sequencing with deadlines the problem is stated as below. There are n jobs to be processed on a machine. Each job i has a deadline $d_i \geq 0$ and profit $p_i \geq 0$. Pi is earned iff the job is completed by its deadline. The job is completed if it is processed on a machine for unit time. Only one machine is available for processing jobs. Only one job is processed at a time on the machine.

A given Input set of jobs 1,2,3,4 have sub sets $2^n$ so $2^4$ $^{=16}$

It can be written as {1},{2},{3},{4},{Ø},{1,2},{1,3},{1,4},{2,3},{2,4},{3,4},{1,2,3}, {1,2,4},{2,3,4},{1,2,3,4},{1,3,4} total of 16 subsets

Problem:

n=4 , P=(70,12,18,35) , d=(2,1,2,1)

| Feasible Solution | Processing Sequence | Profit value | Time Line 0 1 2 |
|---|---|---|---|
| 1 | 1 | 70 | |
| 2 | 2 | 12 | |
| 3 | 3 | 18 | |
| 4 | 4 | 35 | |
| 1,2 | 2,1 | 82 | |
| 1,3 | 1,3 /3,1 | 88 | |
| 1,4 | 4,1 | 105 | |
| 2,3 | 3,2 /2,3 | 30 | |
| 3,4 | 4,3/3,4 | 53 | |

We should consider the pair i,j where di <=dj if di>dj we should not consider pair then reverse the order. We discard pair (2, 4) because both having same dead line(1,1) and cannot process same. Time and discarded pairs (1,2,3), (2,3,4), (1,2,4)…etc since processes are not completed within their deadlines. A feasible solution is a subset of jobs J such that each job is completed by its deadline. An optimal solution is a feasible solution with maximum profit value.

**Example**

Let n = 4, $(p_1,p_2,p_3,p_4) = (100,10,15,27)$, $(d_1,d_2,d_3,d_4) = (2,1,2,1)$

| Sr.No. | Feasible Solution | Processing Sequence | Profit value |
|---|---|---|---|
| (i) | (1, 2) | (2, 1) | 110 |
| (ii) | (1, 3) | (1, 3) or (3, 1) | 115 |
| (iii) | (1, 4) | (4, 1) | 127 is the optimal one |
| (iv) | (2, 3) | (2, 3) | 25 |
| (v) | (3, 4) | (4, 3) | 42 |
| (vi) | (1) | (1) | 100 |
| (vii) | (2) | (2) | 10 |
| (viii) | (3) | (3) | 15 |
| (ix) | (4) | (4) | 27 |

**Problem**: n jobs, S = {1, 2… n}, each job i has a deadline $d_i \geq 0$ and a profit $p_i \geq 0$. We need one unit of time to process each job and we can do at most one job each time. We can earn the profit $p_i$ if job i is completed by its deadline.

The optimal solution = {1, 2, 4}.

The total profit = 20 + 15 + 5 = 40.

| i | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $p_i$ | 20 | 15 | 10 | 5 | 1 |
| $d_i$ | 2 | 2 | 1 | 3 | 3 |

**Algorithm**

Step 1: Sort $p_i$ into non-increasing order. After sorting $p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_i$.

Step 2: Add the next job i to the solution set if i can be completed by its deadline. Assign i to time slot [r-1, r], where r is the largest integer such that $1 \leq r \leq d_i$ and [r-1, r] is free.

Step 3: Stop if all jobs are examined. Otherwise, go to step 2.

Time complexity: $O(n^2)$

Example

| I | $p_i$ | $d_i$ | |
|---|---|---|---|
| 1 | 20 | 2 | assign to [1, 2] |
| 2 | 15 | 2 | assign to [0, 1] |
| 3 | 10 | 1 | Reject |

| 4 | 5 | 3 | assign to [2, 3] |
|---|---|---|---|
| 5 | 1 | 3 | Reject |

solution = {1, 2, 4}
total profit = 20 + 15 + 5 = 40

**Greedy Algorithm to Obtain an Optimal Solution**

Consider the jobs in the non increasing order of profits subject to the constraint that the resulting job sequence J is a feasible solution.

In the example considered before, the non-increasing profit vector is

(100 27 15 10) (2 1 2 1)

$p_1$ $p_4$ $p_3$ $p_2$ $d_1$ d4 d3 d2

J = {1} is a feasible one

J = {1, 4} is a feasible one with processing sequence

J = {1, 3, 4} is not feasible

J = {1, 2, 4} is not feasible

J = {1, 4} is optimal

High level description of job sequencing algorithm

```
Procedure greedy job (D, J, n)
// J is the set of n jobs to 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 deadlines)
     then J:= ← J U {i};
   }
}
```

**Greedy Algorithm for Sequencing unit time jobs**

Procedure JS(d,j,n)

// d(i) ≥ 1, 1≤ i ≤ n are the deadlines, n ≥ 1. The jobs are ordered such that

// $p_1 \geq p_2 \geq \ldots\ldots \geq p_n$ .J[i] is the ith job in the optimal solution , i ≤ i ≤ k. Also, at termination d[J[i] ]≤ d [J[i+1]] ,1 ≤ i ≤ k

```
{
d[0]:=J[0]:=0; //initialize and J(0) is a fictious job with d(0) = 0 //
J[1]:=1; //include job 1
K:=1; // job one is inserted into J //
```

```
for i :=2 to n do // consider jobs in non increasing order of pi //
r:=k;
While ((d[J[r]]>d[i]) and (d[J[r]]#r)) do r:=r-1;
If ((d[J[r] ≤  d[i]) and d[i]>r)) then { //insert i into J[]
For q:=k to (r+1) step-1 do j[q+1]:=j[q];
J[r+1]:=i; k:=k+1;
} } return k;
}
```

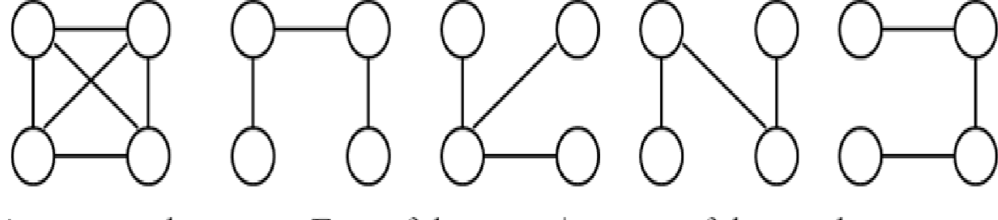
## 3.4 Minimum Cost Spanning Trees

### Spanning trees

Suppose you have a connected undirected graph

- -Connected: every node is reachable from every other node
- -Undirected: edges do not have an associated direction

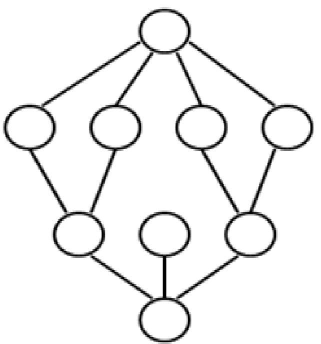
Then a spanning tree of the graph is a connected subgraph in which there are no cycles



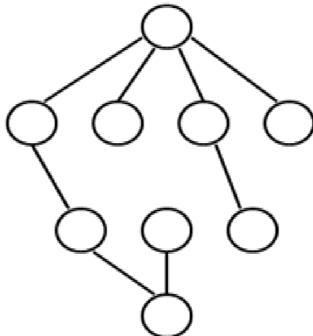
A connected, undirected graph        Four of the spanning trees of the graph

### Finding a spanning tree:

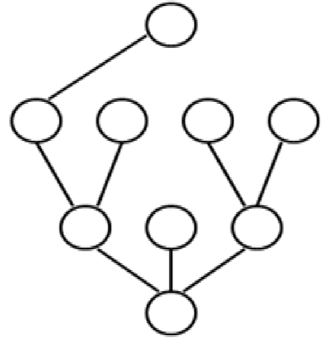
To find a spanning tree of a graph, pick an initial node and call it part of the spanning tree do a search from the initial node: each time you find a node that is not in the spanning tree, add to the spanning tree both the new node *and* the edge you followed to get to it .



An undirected graph



Result of a BFS starting from top



Result of a DFS starting from top

**Minimizing costs**

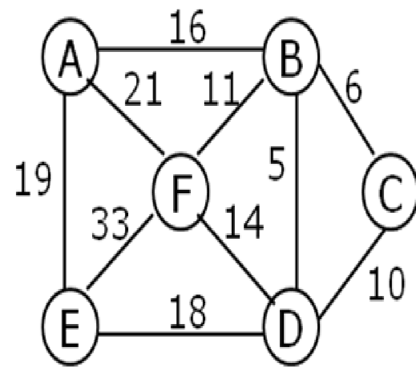
Suppose you want to supply a set of houses (say, in a new subdivision) with:

- electric power
- water
- sewage lines
- telephone lines

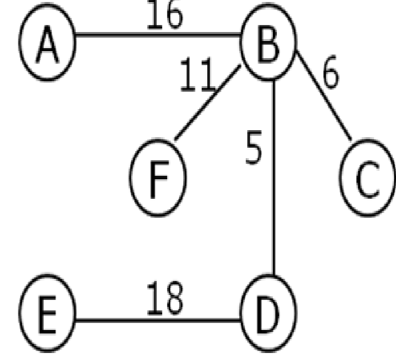
To keep costs down, you could connect these houses with a spanning tree (of, for example, power lines) However, the houses are not all equal distances apart. To reduce costs even further, you could connect the houses with a minimum-cost spanning tree.

**Minimum-cost spanning trees**

Suppose you have a connected undirected graph with a weight (or cost) associated with each edge. The cost of a spanning tree would be the sum of the costs of its edges. A minimum-cost spanning tree is a spanning tree that has the lowest cost.




A connected, undirected graph




A minimum-cost spanning tree

**3.5 Greedy Approach for Prim's and kruskal's algorithm:**

Both Prim's and Kruskal's algorithms are greedy algorithms. The greedy approach works for the MST problem; however, it does not work for many other problems.

**Prim's algorithm:**

```
T = a spanning tree containing a single node s;
E = set of edges adjacent to s;
while T does not contain all the nodes
{
  remove an edge (v, w) of lowest cost from E
  if w is already in T then discard edge (v, w)
  else
 {
  add edge (v, w) and node w to T
   add to E the edges adjacent to w
  }
}
```

An edge of lowest cost can be found with a priority queue. Testing for a cycle is automatic

**Prim's Algorithm:**

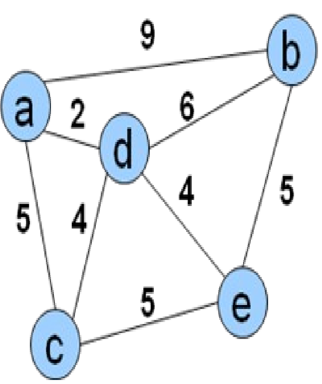
Initialization

a. Pick a vertex $r$ to be the root

b. Set $D(r) = 0$, $parent(r) = null$

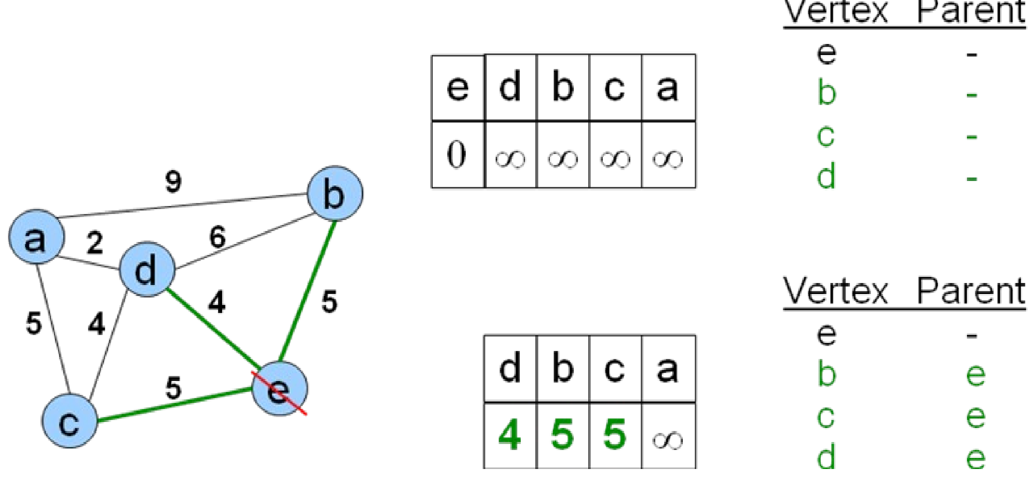
c. For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$

d. Insert all vertices into priority queue $P$, using distances as the keys




| e | a | b | c | d |
|---|---|---|---|---|
| 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

| Vertex | Parent |
|---|---|
| e | - |




The MST initially consists of the vertex *e*, and we update the distances and parent for its adjacent vertices.

| d | b | c | a |
|---|---|---|---|
| 4 | 5 | 5 | ∞ |

| Vertex | Parent |
|---|---|
| e | - |
| b | e |
| c | e |
| d | e |

| a | c | b |
|---|---|---|
| 2 | 4 | 5 |

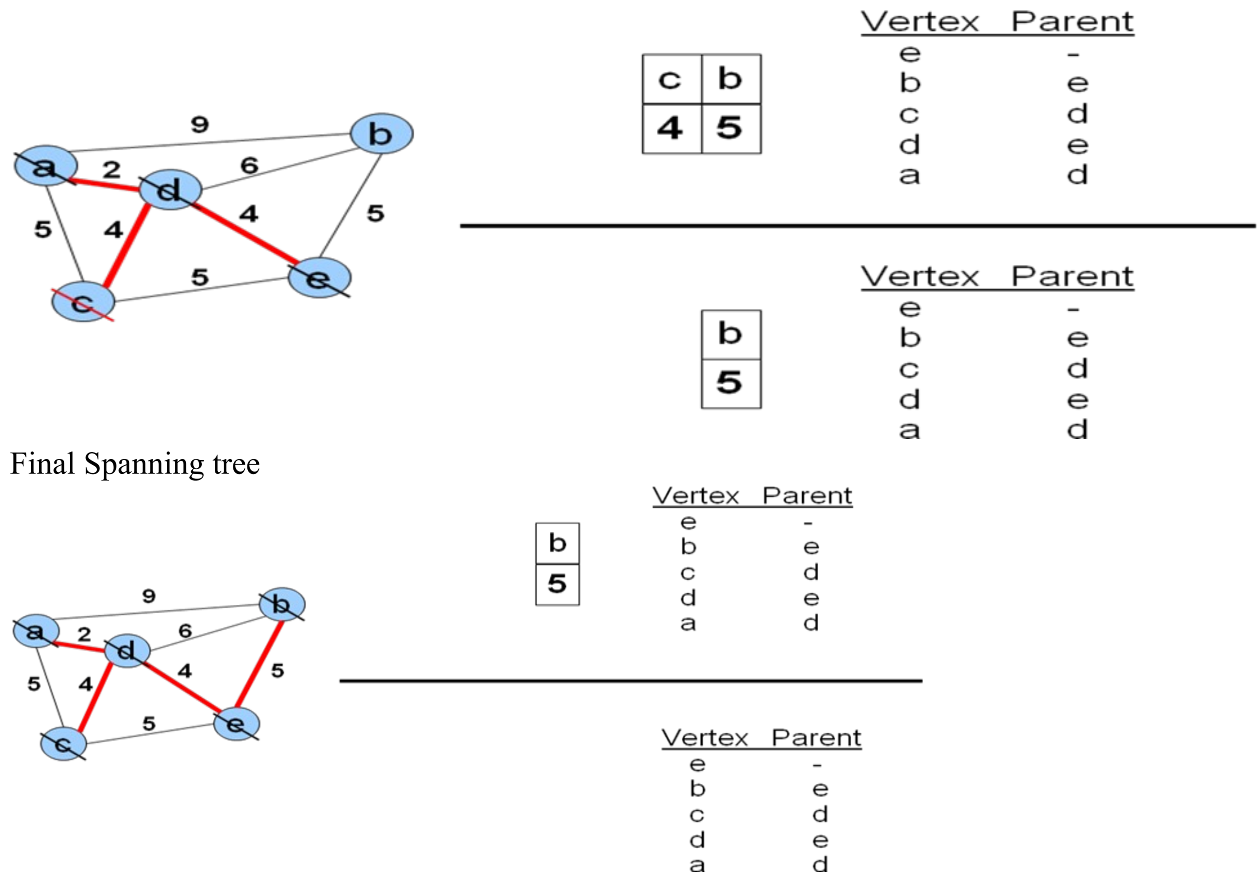
| Vertex | Parent |
|---|---|
| e | - |
| b | e |
| c | d |
| d | e |
| a | d |

| a | c | b |
|---|---|---|
| 2 | 4 | 5 |

| Vertex | Parent |
|---|---|
| e | - |
| b | e |
| c | d |
| d | e |
| a | d |

| c | b |
|---|---|
| 4 | 5 |

| Vertex | Parent |
|---|---|
| e | - |
| b | e |
| c | d |
| d | e |
| a | d |



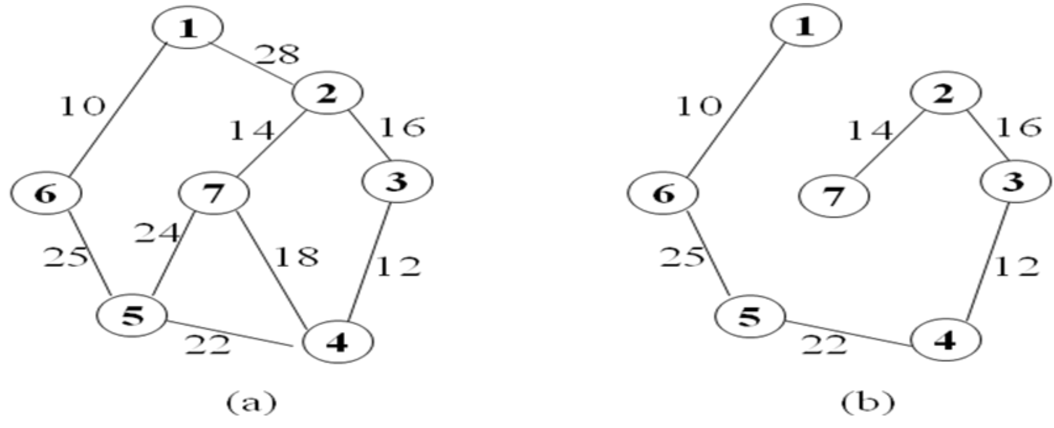

**Running time of Prim's algorithm (without heaps):**

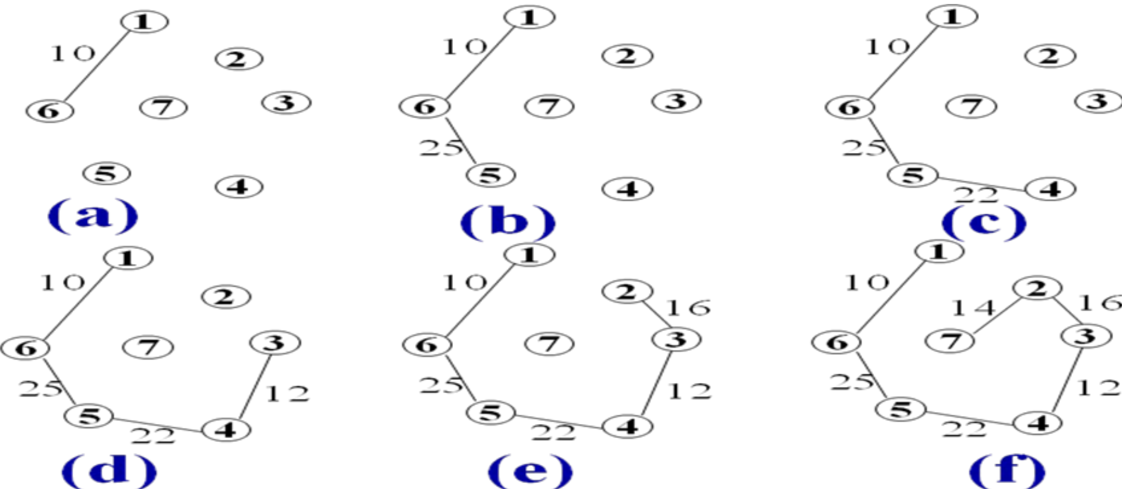
1. Initialization of priority queue (array): O (|*V*|)
2. Update loop: |*V*| calls
   - Choosing vertex with minimum cost edge: O(|*V*|)
   - Updating distance values of unconnected vertices: each edge is considered only once during entire execution, for a total of O(|*E*|) updates

3. Overall cost without heaps:

**Minimum-cost Spanning Trees**

Example of **MCST**: Finding a spanning tree of G with minimum cost



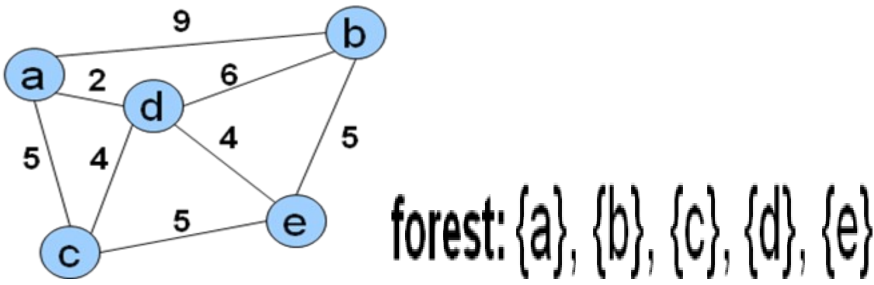



**Prim's Algorithm Invariant:**

At each step, we add the edge $(u,v)$ s.t. the weight of $(u,v)$ is minimum among all edges where $u$ is in the tree and $v$ is not in the tree. Each step maintains a minimum spanning tree of the vertices that have been included thus far. When all vertices have been included, we have a MST for the graph.

**Another Approach:**

Create a forest of trees from the vertices. Repeatedly merge trees by adding "safe edges" until only one tree remains. A "safe edge" is an edge of minimum weight which does not create a cycle.




**Kruskal's algorithm:**

T = empty spanning tree;

E = set of edges;

N = number of nodes in graph;

while T has fewer than N - 1 edges {

remove an edge (v, w) of lowest cost from E

if adding (v, w) to T would create a cycle

then discard (v, w)

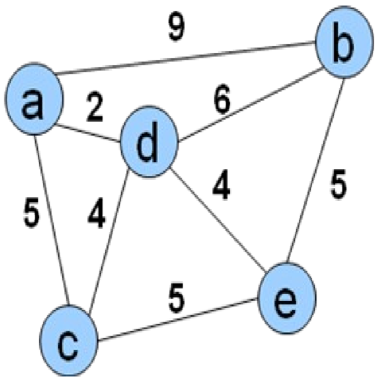
else add (v, w) to T

}

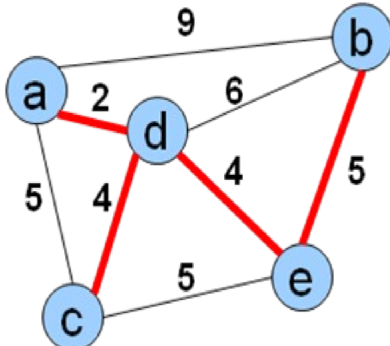
Finding an edge of lowest cost can be done just by sorting the edges
Running time bounded by sorting (or findMin)
$O(|E|\log|E|)$, or equivalently, $O(|E|\ \log|V|)$

Initialization

a. Create a set for each vertex $v \in V$
b. Initialize the set of "safe edges" $A$ comprising the MST to the empty set
c. Sort edges by increasing weight




$F = \{a\}, \{b\}, \{c\}, \{d\}, \{e\}$
$A = \varnothing$
$E = \{(a,d), (c,d), (d,e), (a,c),$
$(b,e), (c,e), (b,d), (a,b)\}$

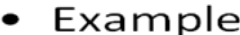


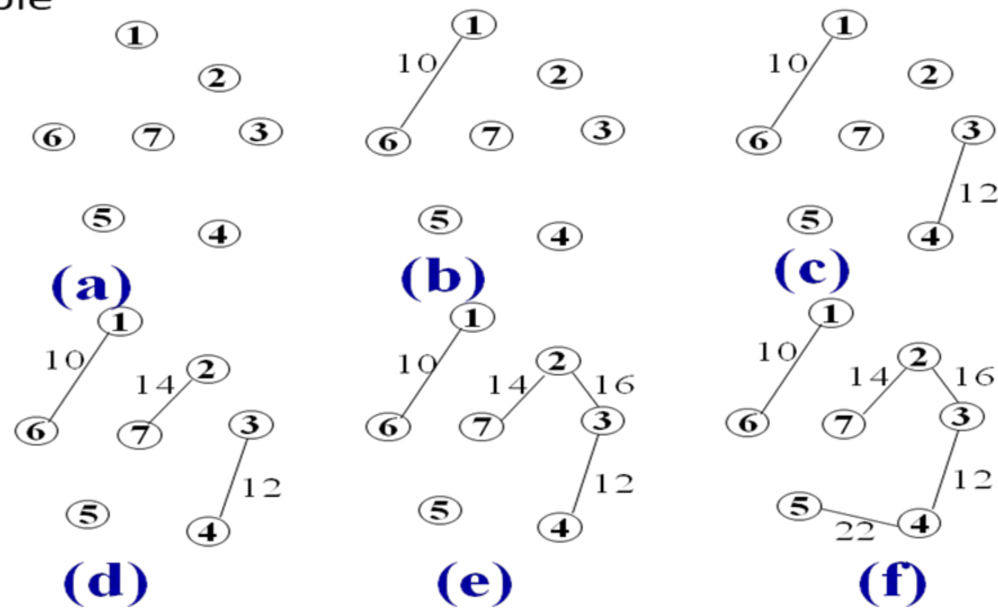

$E = \{(a,d), (c,d), (d,e), (a,c),$
$(b,e), (c,e), (b,d), (a,b)\}$

| Forest | A |
| --- | --- |
| {a}, {b}, {c}, {d}, {e} | ∅ |
| {a,d}, {b}, {c}, {e} | {(a,d)} |
| {a,d,c}, {b}, {e} | {(a,d), (c,d)} |
| {a,d,c,e}, {b} | {(a,d), (c,d), (d,e)} |
| {a,d,c,e,b} | {(a,d), (c,d), (d,e), (b,e)} |

**Kruskal's algorithm Invariant**

After each iteration, every tree in the forest is a MST of the vertices it connects. Algorithm terminates when all vertices are connected into one tree.




### 3.6 Optimal Merge Patterns

**Problem**

Given n sorted files, find an optimal way (i.e., requiring the fewest comparisons or record moves) to pair wise merge them into one sorted file. It fits ordering paradigm.

**Example**

Three sorted files ($x_1$, $x_2$, $x_3$) with lengths (30, 20, 10)

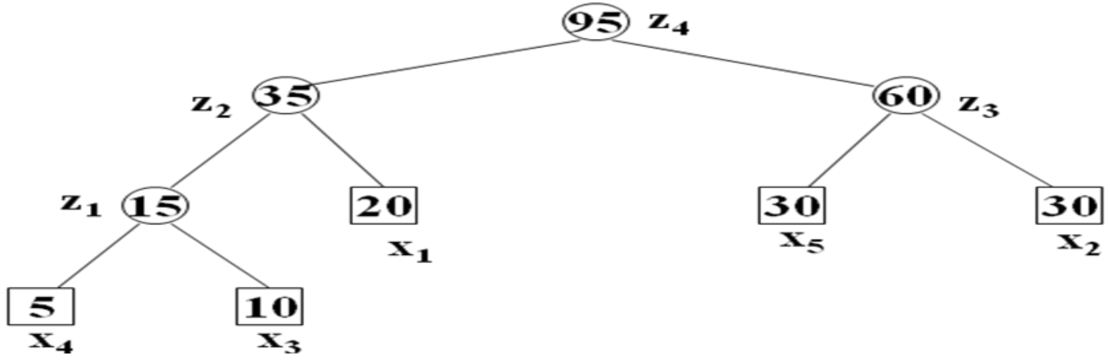
Solution 1: merging $x_1$ and $x_2$ (50 record moves), merging the result with $x_3$ (60 moves) → total 110 moves

Solution 2: merging $x_2$ and $x_3$ (30 moves), merging the result with $x_1$ (60 moves) → total 90 moves

The solution 2 is better.

A greedy method (for 2-way merge problem)

At each step, merge the two smallest files. e.g., five files with lengths (20, 30, 10, 5, 30).



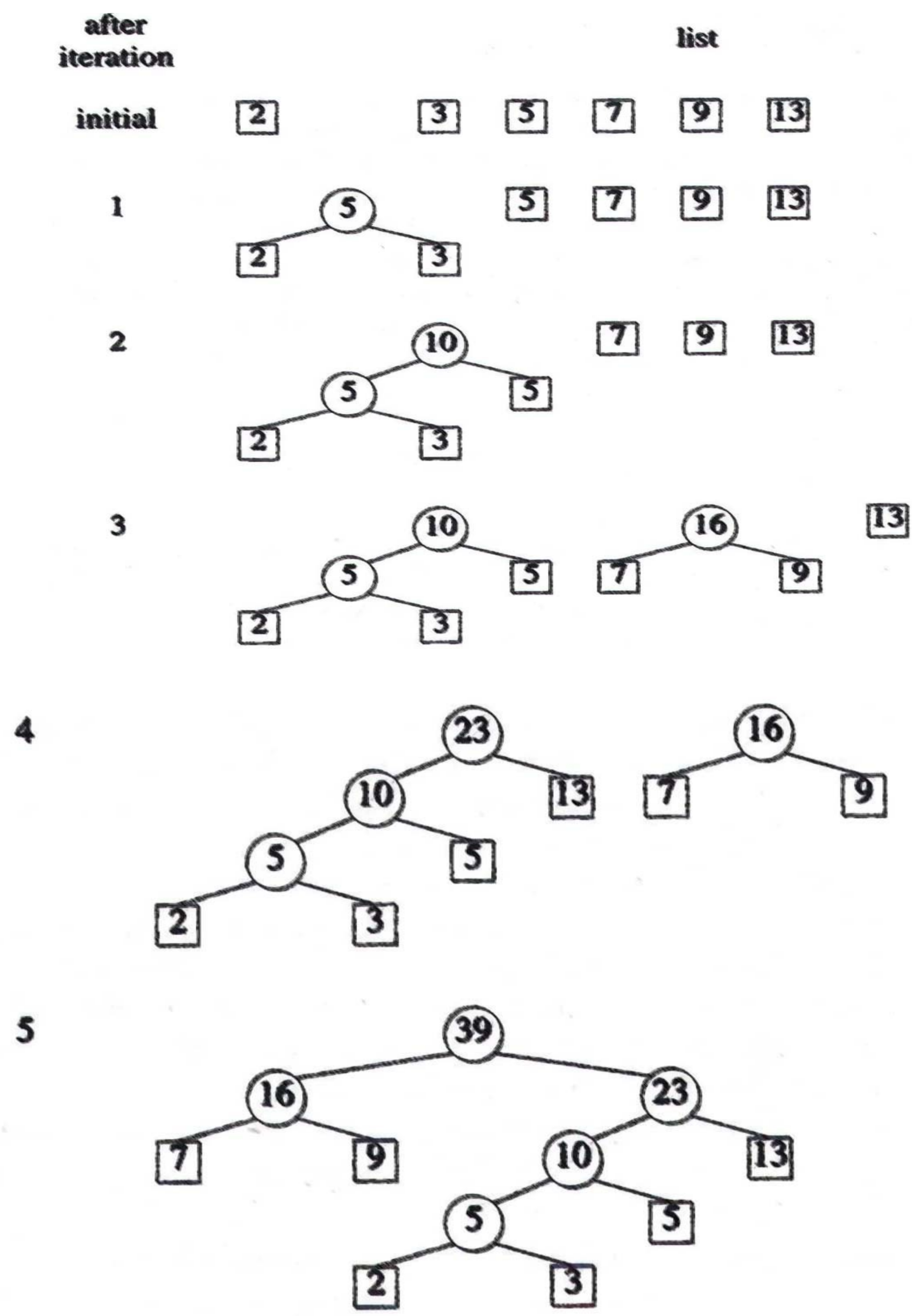

Total number of record moves = weighted external path length

The optimal 2-way merge pattern = binary merge tree with minimum weighted external path length

Algorithm

```
 struct treenode
 {
       struct treenode *lchild, *rchild;
       int weight;
 };
 typedef struct treenode Type;
 Type *Tree(int n)
 //    list is a global list of n single node
 //    binary trees as described above.
 {
       for (int i=1; i<n; i++) {
           Type *pt = new Type;
           // Get a new tree node.
           pt -> lchild = Least(list); // Merge two trees with
           pt -> rchild = Least(list); // smallest lengths.
           pt -> weight = (pt->lchild)->weight
                   + (pt->rchild)->weight;
           Insert(list, *pt);
       }
       return (Least(list)); // Tree left in l is the merge tree.
 }
```

**Example**




**Time Complexity**

If list is kept in non-decreasing order: O ($n^2$)
If list is represented as a min heap: O (n log n)

**3.7 Optimal Storage on Tapes**

There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length $L_i$. Assume the tape is initially positioned at the front. If the programs are stored in the order $I = i_1, i_2 \ldots i_n$, the time $t_j$ needed to retrieve program $i_j$

$$t_j = \sum_{k=1}^{j} L_i \frac{1}{n} \sum_{j=1}^{n} t_j$$

If all programs are retrieved equally often, then the mean retrieval time (MRT) =this problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing

$$D(I) = \sum_{j=1}^{n} \sum_{k=1}^{j} L_{i_k}$$

Example

n=3 (l1, l2, l3) = (5, 10, 3) 3! =6 total combinations

L1 l2 l3 = l1+ (l1+l2) + (l1+l2+l3) = 5+15+18 = 38/3=12.6

n 3

L1 l3 l2 = l1 + (l1+l3) + (l1+l2+l3) = 5+8+18 = 31/3=10.3

n 3

L2 l1 l3 = l2 + (l2+l1) + ( l2+l1+l3) = 10+15+18 = 43/3=14.3

n 3

L2 l3 l1 = 10+13+18 = 41/3=13.6

3

L3 l1 l2 = 3+8+18 = 29/3=9.6 min

3

L3 l2 l1 = 3+13+18 = 34/3=11.3 min

3 permutations at (3, 1, 2)

**Example**

n = 4, $(p_1, p_2, p_3, p_4)$ = (100, 10, 1 5, 27) $(d_1, d_2, d_3, d_4)$ = (2, 1, 2, 1)

| | Feasible solution | Processing sequence | value |
|---|---|---|---|
| 1 | (1,2) | 2,1 | 110 |
| 2 | (1,3) | 1,3 or 3, 1 | 115 |
| 3 | (1,4) | 4, 1 | 127 |
| 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 |
|---|---|---|---|

**Example**

Let n = 3, ($L_1$, $L_2$, $L_3$) = (5, 10, 3). 6 possible orderings. The optimal is 3, 1, 2

| Ordering I | d(I) |
|---|---|
| 1,2,3 | 5+5+10+5+10+3 = 38 |
| 1,3,2 | 5+5+3+5+3+10 = 31 |
| 2,1,3 | 10+10+5+10+5+3 = 43 |
| 2,3,1 | 10+10+3+10+3+5 = 41 |
| 3,1,2 | 3+3+5+3+5+10 = 29 |
| 3,2,1, | 3+3+10+3+10+5 = 34 |

**3.8 TVSP (Tree Vertex Splitting Problem)**

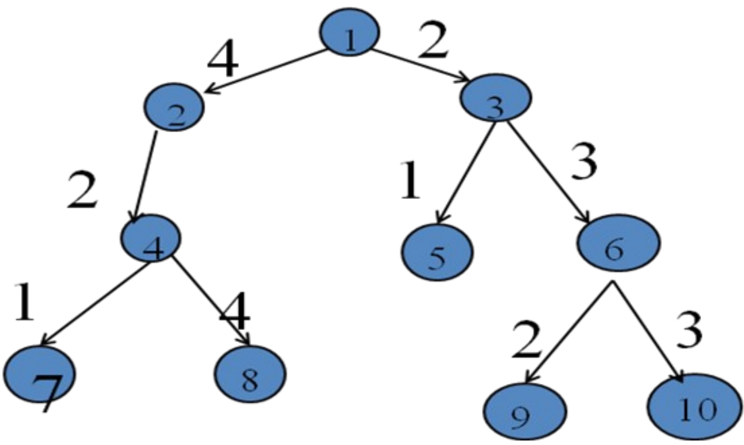
Let T= (V, E, W) be a directed tree. A weighted tree can be used to model a distribution network in which electrical signals are transmitted. Nodes in the tree correspond to receiving stations & edges correspond to transmission lines. In the process of transmission some loss is occurred. Each edge in the tree is labeled with the loss that occurs in traversing that edge. The network model may not able tolerate losses beyond a certain level. In places where the loss exceeds the tolerance value boosters have to be placed. Given a networks and tolerance value, the TVSP problem is to determine an optimal placement of boosters. The boosters can only placed at the nodes of the tree.

d (u) = Max { d(v) + w(Parent(u), u)}

d(u) – delay of node      v-set of all edges & v belongs to child(u)

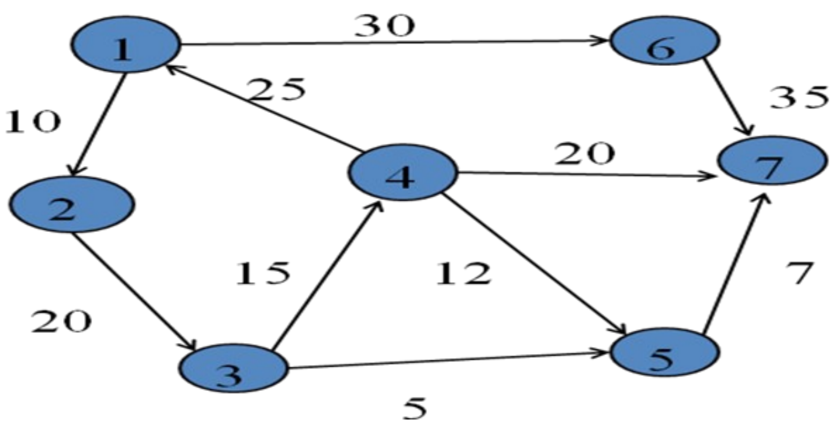
$\delta$ tolerance value

TVSP (Tree Vertex Splitting Problem)



If d (u)>= $\delta$ than place the booster.
d (7)= max{0+w(4,7)}=1
d (8)=max{0+w(4,8)}=4
d (9)= max{0+ w(6,9)}=2
d (10)= max{0+w(6,10)}=3 d(5)=max{0+e(3.3)}=1
d (4)= max{1+w(2,4), 4+w(2,4)}=max{1+2,4+3}=6> $\delta$ ->booster
d (6)=max{2+w(3,6),3+w(3,6)}=max{2+3,3+3}=6> $\delta$->booster
d (2)=max{6+w(1,2)}=max{6+4)=10> $\delta$->booster
d (3)=max{1+w(1,3), 6+w(1,3)}=max{3,8}=8> $\delta$ ->booster
Note: No need to find tolerance value for node 1 because from source only power is transmitting.

### 3.9 Single-source Shortest Paths




Let G=(V,E) be a directed graph and a main function is C(e)(c=cost, e=edge) for the edges of graph 'G' and a source vertex it will represented with $V_0$ the vertices represents cities and weights represents distance between 2 cities. The objective of the problem find shortest path from source to destination. The length of path is defined to be sum of weights of edges on the path. S[i] =T if vertex i present in set's'. S[i] =F if vertex i is not present in set's'

**Formula**

Min {distance[w],distance[u]+cost[u, w]}
u-recently visited node w-unvisited node

Step-1 s[1]

s[1]=T dist[2]=10
s[2]=F dist[3]=$\alpha$
s[3]=F dist[4]= $\alpha$
s[4]=F dist[5]= $\alpha$
s[5]=F dist[6]= 30
s[6]=F dist[7]= $\alpha$
S[7]=F

Step-2 s[1,2] the visited nodes

W={3,4,5,6,7} unvisited nodes

U={2} recently visited node

s[1]=T w=3

s[2]=T dist[3]=α

s[3]=F min {dist[w], dist[u]+cost(u, w)}

s[4]=F min {dist[3], dist[2]+cost(2,3)}

s[5]=F min{α, 10+20}= 30

s[6]=F w=4 dist[4]= α

S[7]=F min{dist(4),dist(2)+cost(2,4)}

min{α,10+ α}= α

W=5 dist[5]= α min{dist(5),dist(2)+cost(2,5)}

min{α,10+ α}= α

W=6 dist[6]=30

Min{dist(6), dist(2)+cost(2,6)}=min{30,10+ α}=30

W=7, dist(7)= α min{dist(7),dist(2)+cost(2,7)}

min{α,10+ α}= α let min. cost is 30 at both 3 and 6 but

Recently visited node 2 have only direct way to 3, so consider 3 is min cost node from 2.

Step-3 w=4,5,6,7

s[1]=T s={1,2,3} w=4 ,dist[4]= α

s[2]=T min{dist[4],dist[3]+cost(3.4)}=min{α,30+15}=45

s[3]=T w=5, dist[5]= α min{dist(5), dist(3)+cost(3,5)}

s[4]=F min{α,30+5}=35 similarity we obtain

s[5]=F w=6, dist(6)=30 w=7 ,dist[7]= α so min cost is 30 at w=6 but

s[6]=F no path from 3 so we consider 5 node so visited nodes 1,2,3, 5

S[7]=F

Step-4 w=4,6,7 s={1,2,3,5}

s[1]=T w=4, dist[4]=45 min {dist[4], dist[5]+cost(5,4)}

s[2]=T min{45,35+ α}=45

s[3]=T w=6,dist[6]=30 min{dist[6],dist[5]+cost(5,6)}

s[4]=F min{30, 35+ α}=30

s[5]=T w=7,dist[7]= α min{dist[7],dist[5]+cost(5,7)}

s[6]=F min{α, 35+7}=42

S[7]=F here min cost is 30 at 6 node but there is no path from 5 yo 6, so we consider 7 , 1,2,3,5,7 nodes visited.

Therefore the graph traveled from source to destination

Single source shortest path is drawn in next slide.

**Design of greedy algorithm**

Building the shortest paths one by one, in non-decreasing order of path lengths

e.g., 1→4: 10

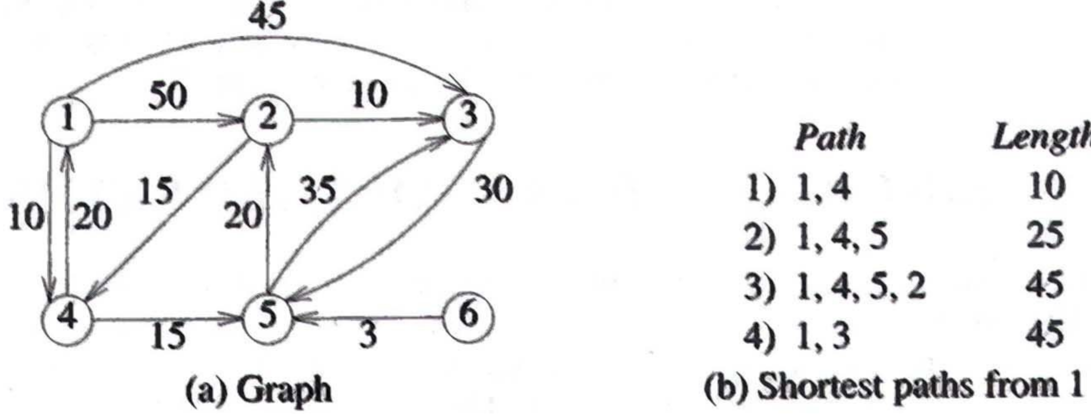
1→4→5: 25

…

We need to determine 1) the next vertex to which a shortest path must be generated and 2) a shortest path to this vertex.

Notations

*S* = set of vertices (including $v_0$) to which the shortest paths have already been generated
*Dist (w)* = length of shortest path starting from $v_0$, going through only those vertices that are in *S*, and ending at *w*.
Three observations
If the next shortest path is to vertex *u*, then the path begins at $v_0$, ends at *u*, and goes through only those vertices that are in *S*. The destination of the next path generated must be that of vertex *u* which has the minimum distance, *dist (u),* among all vertices not in *S*.
Having selected a vertex *u* as in observation 2 and generated the shortest $v_0$ to *u* path, vertex *u* becomes a member of *S*.

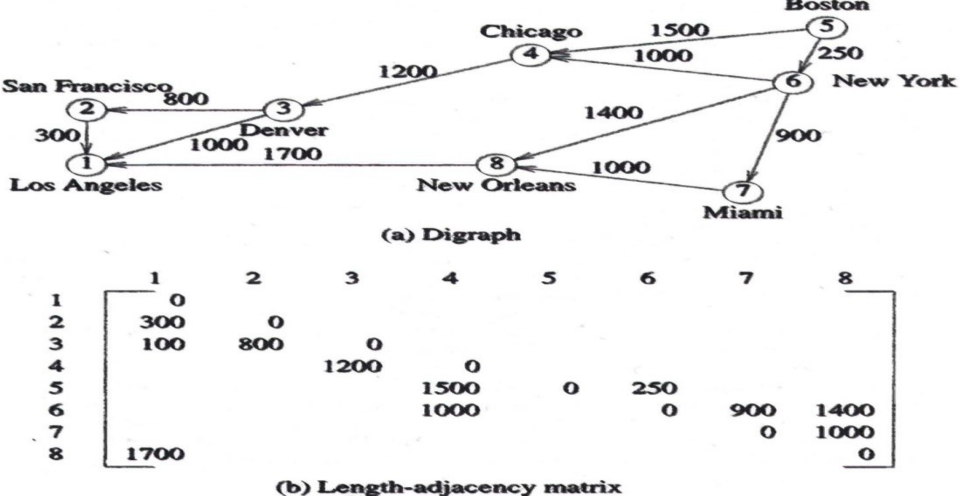



**DIJKSTRA'S Shortest Path Algorithm**

Procedure SHORT-PATHS (v, cost, Dist, n)
// Dist (j) is the length of the shortest path from v to j in the //graph G with n vertices; Dist (v)= 0 //
Boolean S (1:n); real cost (1:n,1:n), Dist (1:n); integer u, v, n, num, i, w
// S (i) = 0 if i is not in S and s(i) =1 if it is in S//
// cost (i, j) = +α if edge (i, j) is not there//
// cost (i,j) = 0 if i = j; cost (i, j) = weight of < i, j >
// for i←1 to do // initialize S to empty
// S(i) ←0; Dist (i)← cost(v, i)
Repeat
// initially for no vertex shortest path is available
// S (v)←1; dist(v)←0// Put v in set S //
for num←2 to n-1 do // determine n-1 paths from// //vertex v //
choose u such that Dist (u)=min{dist(w)} and S(w)=0
S (u)←1 // Put vertex u in S //
Dist (w)←min (dist(w),Dist(u) + cost (u, w))
Repeat

repeat
end SHORT - PATHS
Overall run time of algorithm is O ((n+|E|) log n)

**Example:**



(a) Digraph

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | | | | | | | |
| 2 | 300 | 0 | | | | | | |
| 3 | 100 | 800 | 0 | | | | | |
| 4 | | | 1200 | 0 | | | | |
| 5 | | | | 1500 | 0 | 250 | | |
| 6 | | | | 1000 | | 0 | 900 | 1400 |
| 7 | | | | | | | 0 | 1000 |
| 8 | 1700 | | | | | | | 0 |

(b) Length-adjacency matrix

| Iteration | S | Vertex selected | Distance LA [1] | SF [2] | DEN [3] | CHI [4] | BOST [5] | NY [6] | MIA [7] | NO [8] |
|---|---|---|---|---|---|---|---|---|---|---|
| Initial | -- | ---- | +∞ | +∞ | +∞ | 1500 | 0 | 250 | +∞ | +∞ |
| 1 | {5} | 6 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 2 | {5,6} | 7 | +∞ | +∞ | +∞ | 1250 | 0 | 250 | 1150 | 1650 |
| 3 | {5,6,7} | 4 | +∞ | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 4 | {5,6,7,4} | 8 | 3350 | +∞ | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 5 | {5,6,7,4,8} | 3 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| 6 | {5,6,7,4,8,3} | 2 | 3350 | 3250 | 2450 | 1250 | 0 | 250 | 1150 | 1650 |
| | {5,6,7,4,8,3,2} | | | | | | | | | |

********

## Chapter-4
## Dynamic programming

### 4.1 The General Method

**Dynamic Programming:** is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.

**The shortest path**

To find a shortest path in a multi-stage graph