# Greedy Algorithms

# Constrained Optimization Problems

A problem in which some function of certain variables (called the optimization or *objective function*) is to be optimized (usually minimized or maximized) subject to some *constraints*.

**Types of solutions:**

- **Feasible solution**: Any assignment of values to the variables that satisfies the given constraints.

- **Optimal solution**: A feasible solution that optimizes the objective function.

# Greedy Algorithms

- At each step in the algorithm, one of several choices can be made.

- **Greedy Strategy**: make the choice that is the best at the moment.

- After making a choice, we are left with **one subproblem** to solve.

- The solution is created by making a sequence of **locally optimal** choices.

# Greedy Algorithms: Optimality Conditions

**Greedy Choice property:**
A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

**Optimal Substructure:**
An optimal solution to the problem contains within it optimal solutions to subproblems.

## Greedy Algorithms: Examples

- Prim's algorithm: Each step, include a new edge into the set $A$. Greedy criterion: select the **minimum-weight** edge connecting a vertex inside $A$ and a vertex outside $A$ (i.e., select a vertex that has smallest $key$ value).

- Kruskal's algorithm: Each step, include a new edge into the set $A$. Greedy criterion: select the **minimum-weight** edge connecting two trees in $A$.

- Dijkstra's algorithm: Each step, include a new vertex into the set $S$. Greedy criterion: select the vertex with **smallest** d[u] value (i.e., the vertex that is closest to the source $s$).

# Optimization Problems

- Subset Paradigm
  - Selecting a subset of input based on some optimization measure
  - Examples – Container loading, Knapsack Filling, Job sequencing with deadlines, minimum cost spanning trees
- Ordering Paradigm
  - Considering the inputs in some order
  - Making decisions using an optimization criterion that can be computed using decisions already made
  - Examples – Optimal storage on tapes, Optimal merge pattern, Single souce shortest path

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

**Algorithm 4.1** Greedy method control abstraction for the subset paradigm

# Knapsack Problem: Formal Description

- Input: $n$ objects and a knapsack.
- Each object $i$ has a weight $w_i$, a value $p_i$ and the knapsack has a capacity $m$.
- A fraction of object $x_i, 0 \leq x_i \leq 1$ yields a profit of $p_i \cdot x_i$.
- Objective is to obtain a filling that maximizes the profit, under the weight constraint of $m$.
- **Optimization Problem:** find $x_1, x_2, ..., x_n$, such that:

$$\begin{cases} \text{maximize:} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{subject to:} & \sum_{i=1}^{n} w_i \cdot x_i \leq m \\ & \text{and } 0 \leq x_i \leq 1, 1 \leq i \leq n \end{cases}$$

## Two Observations

**Lemma 1** In case $\sum_{i=1}^{n} w_i \leq m$, then $x_i = 1, 1 \leq i \leq n$ is an optimal solution.

**Lemma 2** In case $\sum_{i=1}^{n} w_i \geq m$, all optimal solutions will fit the knapsack exactly.

# Problem Instance

$n = 3, m = 20, P = (25, 24, 15)$ and $W = (18, 15, 10)$.

Solution 1: $x_1 = 0.5, x_2 = \frac{1}{3}, x_3 = \frac{1}{4}$

Constraint $\Rightarrow$ $\underbrace{\sum w_i \cdot x_i = 16.5}_{\text{a feasible solution}} \Rightarrow$ Total profits $= 24.25$ (0.5 x 25 + 1/3 x 24 + ¼ x 15)

Solution 2: $x_1 = 0.0, x_2 = 1.0, x_3 = \frac{1}{2}$

$\underbrace{\sum w_i \cdot x_i = 20}_{\text{a feasible solution}} \Rightarrow$ Total profits $= 31.5$ $\Leftarrow$ Optimal Solution

# Possible Greedy Strategies

$n = 3, m = 20, P = (25, 24, 15)$ and $W = (18, 15, 10)$.

**Strategy 1:** Pick the max-value object first.
Choose the object in nonincreasing order of value.

$$x_1 = 1,\ x_2 = \frac{2}{15},\ x_3 = 0 \Rightarrow \sum p_i \cdot x_i = 28.2$$

**Strategy 2:** Pick the lightest object first.
Choose the object in nondecreasing order of weight.

$$x_3 = 1,\ x_2 = \frac{2}{3},\ x_1 = 0 \Rightarrow \sum p_i \cdot x_i = 31$$

$n = 3, m = 20, P = (25, 24, 15)$ and $W = (18, 15, 10)$.

## Pick the object with the maximum value per pound

**Strategy 3:** Choose the object in nonincreasing order of $\frac{p_i}{w_i}$

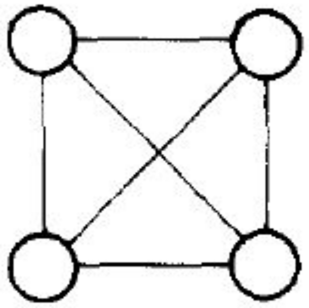
$$\frac{p_i}{w_i} = (\frac{25}{18}, \frac{24}{15}, \frac{15}{10}) = (1.39, 1.60, 1.5)$$

so $x_2 = 1, x_3 = \frac{1}{2}, x_1 = 0 \Rightarrow \sum p_i \cdot x_i = 31.5$

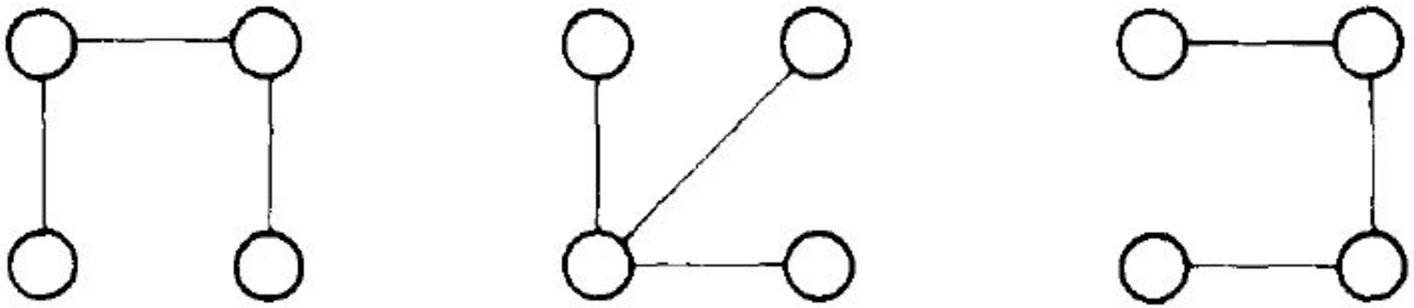
# Greedy Knapsack

```
void GreedyKnapsack(float m, int n)
// p[1..n] and w[1..n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] ≥ p[i+1]/w[i+1]. m is the knapsack
// capacity and x[1..n] is the solution vector.

    for i := 1 to n   x[i] = 0.0;       // initialize x

    U := m;
    for i := 1 to n
        if (w[i] > U) break;
        x[i] := 1.0;                  // put the whole object in
        U := U - w[i];

    if (i ≤ n) x[i] := U/w[i];        // the last object to be put in
```

# Spanning Tree

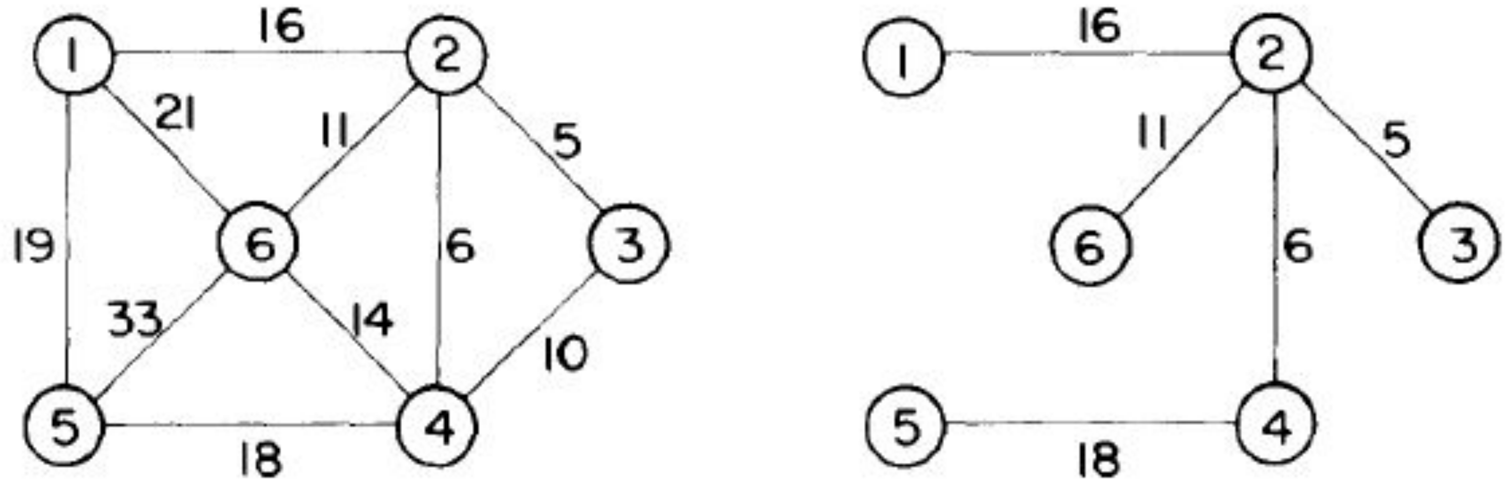


Graph



Spanning Tree

# Minimum Cost Spanning Tree




Objective function

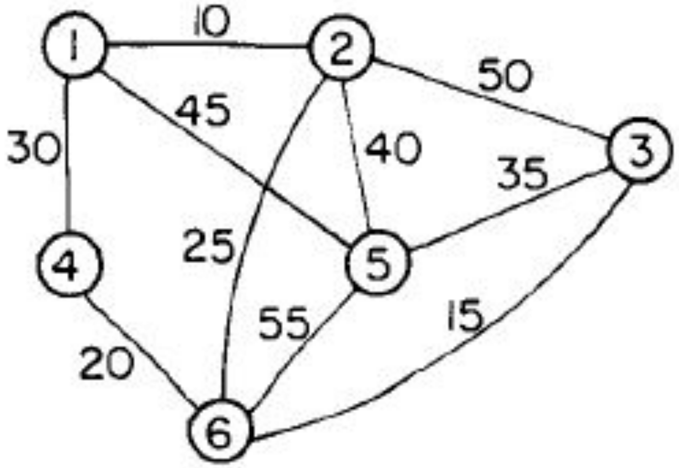
Minimize sum of cost of the edges in the MST

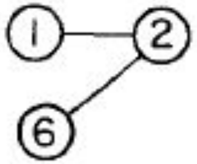
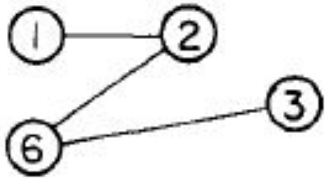
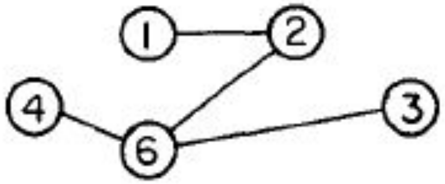
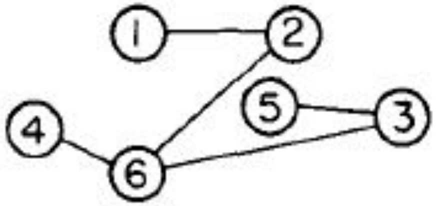
Constraints - No Cycle

# Prim’s Algorithm

- Minimum cost spanning tree is build edge by edge.
- Optimization criteria is t o choose an edge that results in a minimum increase in the sum of costs of the edges so far included.
- The set of edges so far selected form a tree.

# Prim’s Algorithm




| Edge | Cost | Spanning tree |
|---|---|---|
| (1,2) | 10 |  |
| (2,6) | 25 |  |
| (3,6) | 15 |  |
| (6,4) | 20 |  |
| (1,4) | reject | |
| (3,5) | 35 |  |

Cost

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 | 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

T

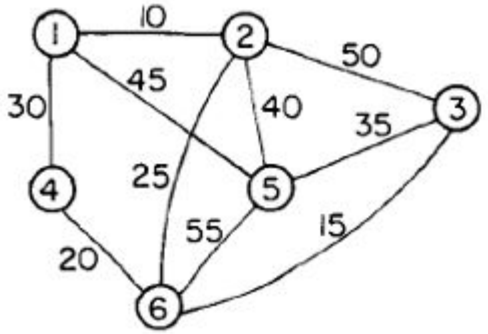
| | |
|---|---|
| 1 | 2 |
| | |
| | |
| | |
| | |


Near

| | |
|---|---|
| 1 | ~~1~~ 0 |
| 2 | ~~2~~ 0 |
| 3 | 2 |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |

(k,l) = (1,2) [min cost edge]

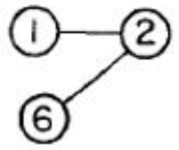
mincost = 10



Cost

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 | T 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

| | |
|---|---|
| 1 | 2 |
| 6 | 2 |
| | |
| | |
| | |




Near

mincost = 10

update Near

| | |
|---|---|
| 1 | ~~1~~ 0 |
| 2 | ~~2~~ 0 |
| 3 | 2 |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |

min (cost(3,2), cost(4,1), cost(5,2),(cost(6,2))

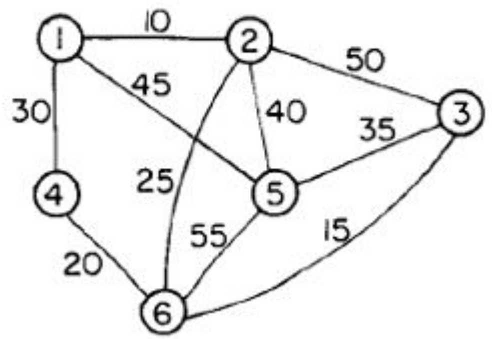
= min(50, 30, 40, 25) = 25

mincost = 10 +25 = 35

add edge (6,2)

update Near array

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | ~~2~~ 6 |
| 4 | ~~1~~ 6 |
| 5 | 2 |
| 6 | ~~2~~ 0 |

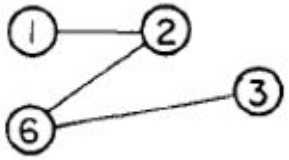


Cost

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 | 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

T

| | |
|---|---|
| 1 | 2 |
| 6 | 2 |
| 3 | 6 |
| | |
| | |




Near

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 6 |
| 4 | 6 |
| 5 | 2 |
| 6 | 0 |

mincost = 35

min (cost(3,6), cost(4,6), cost(5,2))

= min(15, 55, 40) = 15

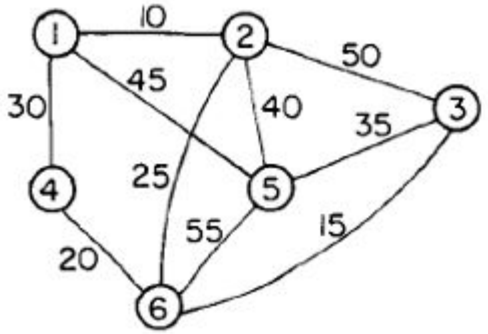
mincost = 35 +15 = 50

add edge (3,6)

update Near array

update Near

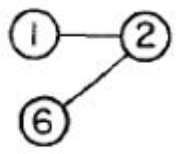
| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | ~~6~~ 0 |
| 4 | ~~6~~ 6 |
| 5 | 2 |
| 6 | 0 |



Cost

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 | T 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

| | |
|---|---|
| 1 | 2 |
| 2 | 6 |
| | |
| | |
| | |




mincost = 10

min (cost(2,3), cost(1,4), cost(2,5),(cost(2,6))

= min(50, 30, 40, 25) = 25

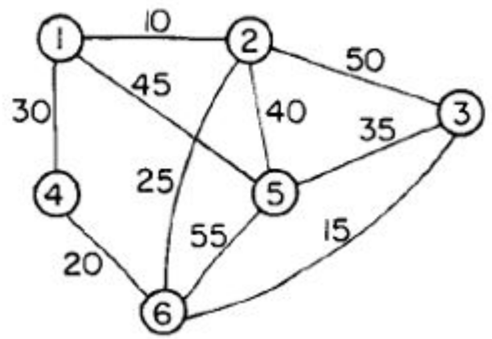
mincost = 10 +25 = 35

add edge (2,6)

update Near array

update Near

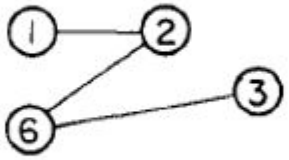
| | | | |
|---|---|---|---|
| 1 | ~~1~~ 0 | 1 | 0 |
| 2 | ~~2~~ 0 | 2 | 0 |
| 3 | 2 | 3 | ~~2~~ 6 |
| 4 | 1 | 4 | ~~1~~ 6 |
| 5 | 2 | 5 | 2 |
| 6 | 2 | 6 | ~~2~~ 0 |



Cost

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 | 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

| | |
|---|---|
| 1 | 2 |
| 2 | 6 |
| 6 | 3 |
| | |
| | |




mincost = 35

update Near

min (cost(6,3), cost(6,4), cost(2,5))

= min(15, 55, 40) = 15

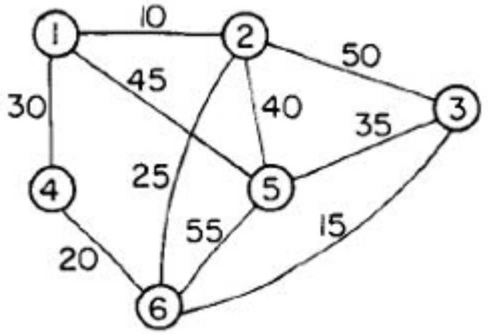
mincost = 35 +15 = 50

add edge (6,3)

update Near array

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 6 |
| 4 | 6 |
| 5 | 2 |
| 6 | 0 |

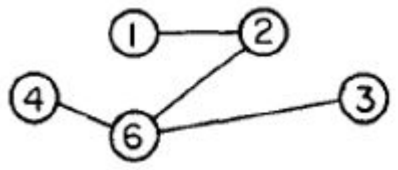
| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | ~~6~~ 0 |
| 4 | ~~6~~ 6 |
| 5 | 2 |
| 6 | 0 |



Cost

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 T | 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

| | |
|---|---|
| 1 | 2 |
| 6 | 2 |
| 3 | 6 |
| 4 | 6 |
| | |




Near

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | 3 |
| 5 | 2 |
| 6 | 0 |

mincost = 50

min (cost(4,6), cost(5,2))

= min(20, 40) = 20

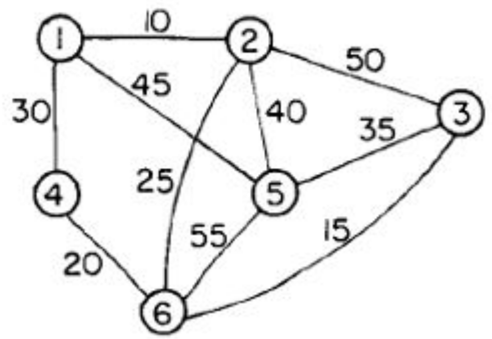
mincost = 50 +20 = 70

add edge (4,6)

update Near array

update Near

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | ~~3~~ 0 |
| 5 | ~~2~~ 3 |
| 6 | 0 |

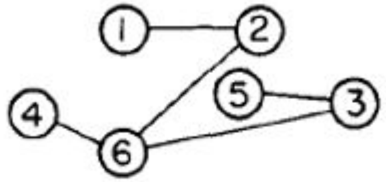


Cost

T

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 | 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

| | |
|---|---|
| 1 | 2 |
| 6 | 2 |
| 3 | 6 |
| 4 | 6 |
| 5 | 3 |




Near

mincost = 70

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | 0 |
| 5 | 3 |
| 6 | 0 |

min (cost(5,3))

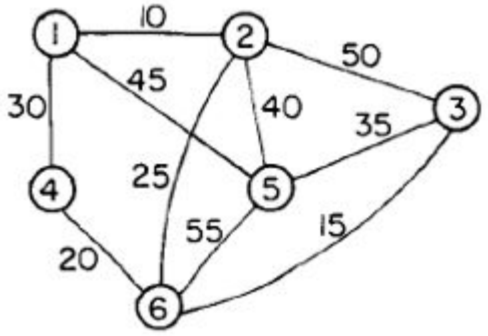
= min(35) = 35

mincost = 70 +35 = 105

add edge (5, 3)

update Near array

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 0 |
| 4 | ~~3~~ 0 |
| 5 | ~~2~~ 0 |
| 6 | 0 |



# PrimsMST(E, cost, n, t)

- n  - number of vertices
- E – set 0f edges
- Cost[1:n, 1:n]  – nxn cost adjacency matrix

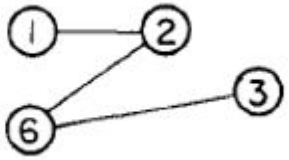
    cost(i, j) = a positive number if edge(i, j) exists

    = ∞ if no edge (i, j) exists
- t(1:n-1, 1:2) – MST    [(t(I,1], t[I,2]) is an edge in MST]
- Minimum cost is returned

Cost

| | | | | | |
|---|---|---|---|---|---|
| 0 | 10 | ∞ | 30 | 45 | ∞ |
| 10 | 0 | 50 | ∞ | 40 | 25 |
| ∞ | 50 | 0 | ∞ | 35 | 15 |
| 30 | ∞ | ∞ | 0 | ∞ | 20 |
| 45 | 40 | 35 | ∞ | 0 | 55 |
| ∞ | 25 | 15 | 20 | 55 | 0 |

| | |
|---|---|
| 1 | 2 |
| 6 | 2 |
| 3 | 6 |
| | |
| | |




Near

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | 6 |
| 4 | 6 |
| 5 | 2 |
| 6 | 0 |

mincost = 35

min (cost(3,6), cost(4,6), cost(5,2))

= min(15, 55, 40) = 15

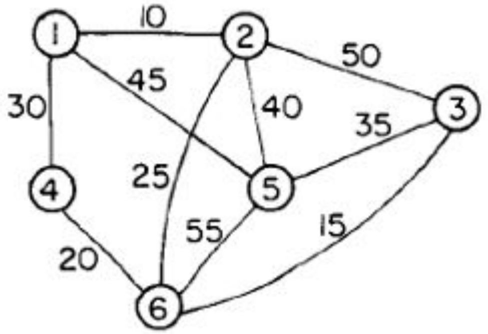
mincost = 35 +15 = 50

add edge (3,6)

update Near array

update Near

| | |
|---|---|
| 1 | 0 |
| 2 | 0 |
| 3 | ~~6~~ 0 |
| 4 | ~~6~~ 6 |
| 5 | 2 |
| 6 | 0 |



# Initialization

Let $(k, l)$ be an edge of minimum cost in $E$;
$mincost := cost[k, l]$;
$t[1, 1] := k$; $t[1, 2] := l$;

T

| 1 | 2 |
|---|---|
| | |
| | |
| | |
| | |

Updation on Near

**for** $i := 1$ **to** $n$ **do** // Initialize near.
    **if** $(cost[i, l] < cost[i, k])$ **then** $near[i] := l$;
    **else** $near[i] := k$;
$near[k] := near[l] := 0$;

Near

| | |
|---|---|
| 1 | ~~1~~ 0 |
| 2 | ~~2~~ 0 |
| 3 | 2 |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |

# Construction of MST

```
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;
}
```

```
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;              O(|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;       Theta(n)
        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          O(n)
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];                              O(n²)
        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]))   O(n)
                then near[k] := j;
    }
    return mincost;
}
```

**Algorithm** Prim's minimum-cost spanning tree algorithm

# Optimal Storage on Tapes

# Optimal Storage on Tapes

- Consider storing n programs of length $l_i$ on a computer tape of length $l$
- Let the programs be stored in the order $I = i_1, i_2, \ldots, i_n$
- Time taken to retrieve program $i_j$ is proportional to $\sum_{1 \le k \le j} l_{i_k}$
- Mean retrieval time(MRT) = (1/n) $\sum_{1 \le k \le j} l_{i_k}$
- Objective – minimize MRT
- Constraint – sum of the lengths of the programs is at most $l$
- Minimizing MRT …t to

  minimize d(I) = $\sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{i_k}$

# Optimal Storage on Tapes

Let $n = 3$ and $(l_1, l_2, l_3) = (5, 10, 3)$. There are $n! = 6$ possible orderings. These orderings and their respective $D$ values are:

| *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 | $3 + 3 + 10 + 3 + 10 + 5 = 34$ |

The optimal ordering is 3,1,2.

Greedy algorithm : store the programs in nondecreasing order of their lengths

# Storage on multiple tapes

If the jobs are initially ordered so that $l_1 \leq l_2 \leq \cdots \leq l_n$, then the first $m$ programs are assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The next $m$ programs will be assigned to tapes $T_0, \ldots, T_{m-1}$ respectively. The general rule is that program $i$ is stored on tape $T_{i \bmod m}$.

Example : Find an optimal placement for 10 programs on three tapes where the program lengths are 12, 5, 8, 32, 7, 5, 18, 26, 4, 11

# Storage on multiple tapes

Example : Find an optimal placement for 10 programs on three tapes where the program lengths are 12, 5, 8, 32, 7, 5, 18, 26, 4, 11

Programs in the order of their lengths : 4, 5, 5, 7, 8, 11, 12, 18, 26, 32

Tape $T_0$ : 4, 7, 12, 32

Tape $T_1$ : 5, 8, 18

Tape $T_2$ : 5, 11, 26

# Storage on Tapes - Algorithm

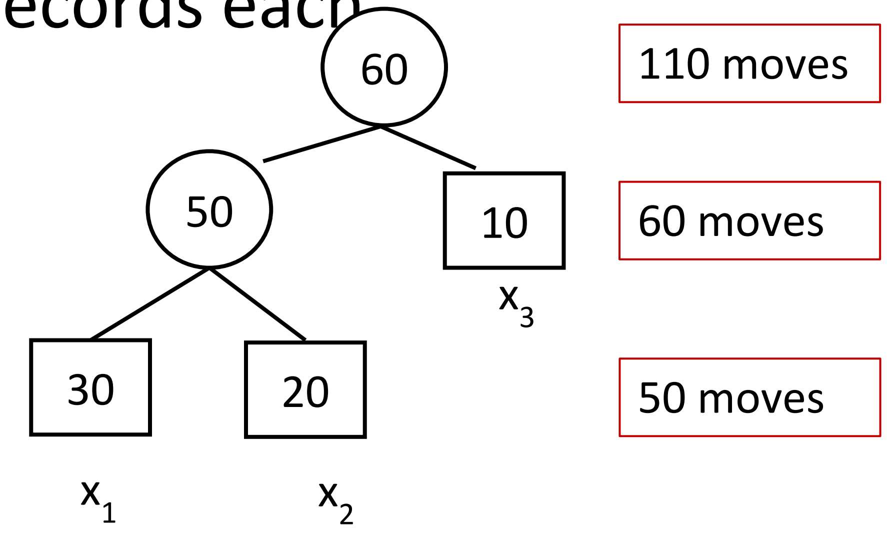
Algorithm store(n, m)

// n – no. of programs and m – number of tapes

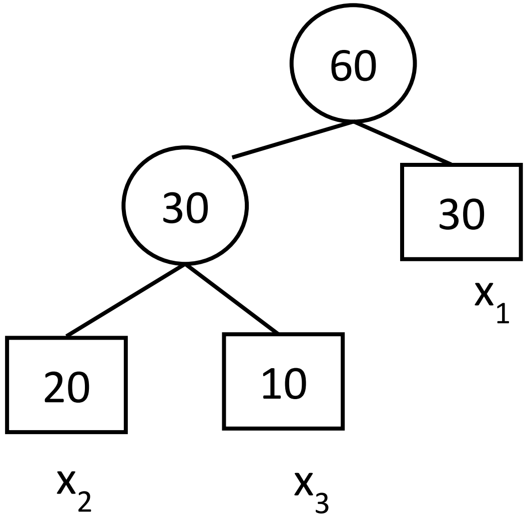
1. j = 0
2. For i = 1 to n do
    1. Print (“Append Program”, i, “to tape”, j)
    2. j = (j + 1) mod m;
3. end

# Optimal Merge Pattern

- When two or more sorted files are merged, merge is accomplished by repeatedly merging sorted files in pairs
- Example : Files $x_1$, $x_2$, $x_3$ are three sorted files of length 30, 20 and 10 records each




Total moves 50 + 60 = 110
Order – merge $x_1$, $x_2$ and with the result merge $x_3$




Total moves 30 + 60 = 90
Order – merge $x_2$, $x_3$ and with the result merge $x_1$

# Greedy Strategy – Two way merge pattern

- At each step merge the two smallest size files together
- Files of smaller length will be merged several times resulting in the minimum increase in the record movements
- If $d_i$ is the distance from the root to external node for file $x_i$ and $q_i$, the length of $x_i$ is then the total number of records moves for this binary merge tree is

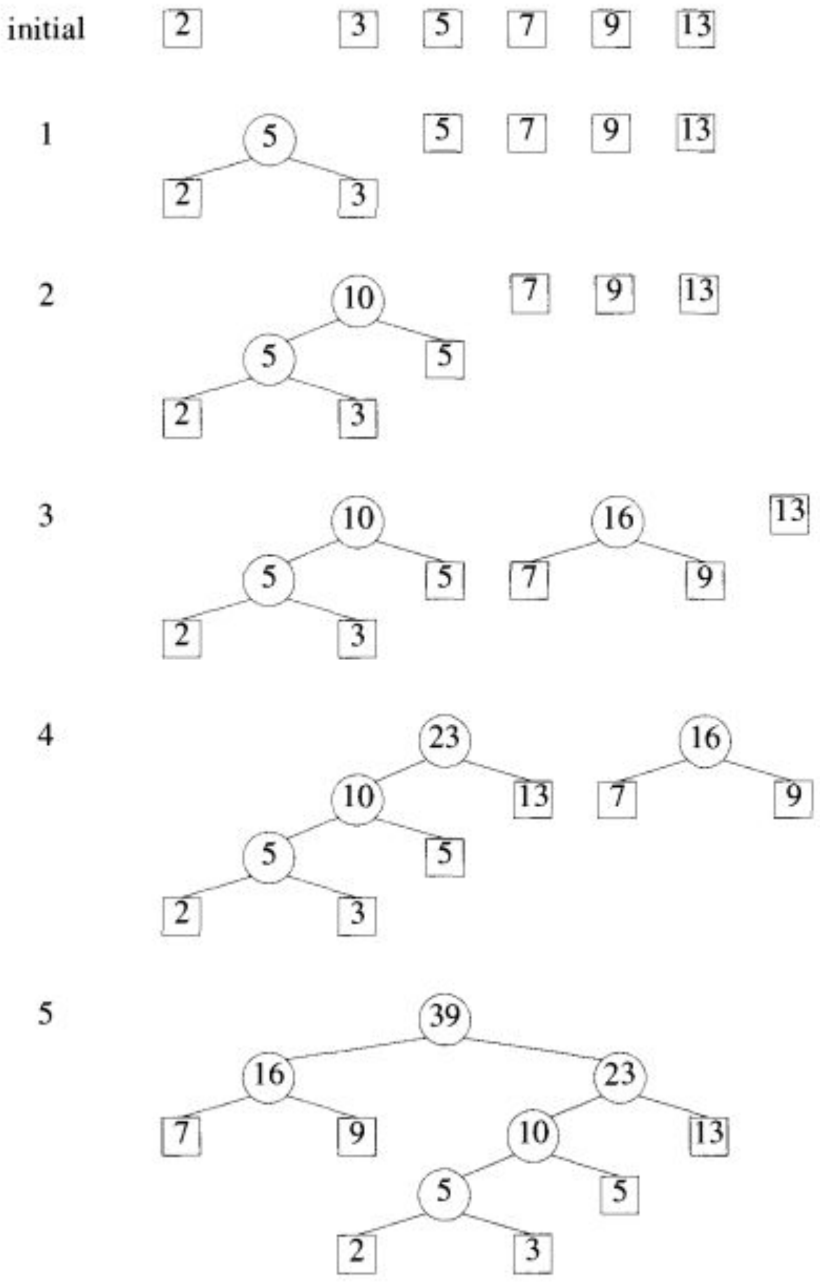
$$\sum_{i=1}^{n} d_i q_i$$

- This sum is called the weighted external path length of the tree.
- An optimal two-way merge pattern corresponds to a binary merge tree with minimum weighted external path length

# Algorithm to generate two way merge pattern

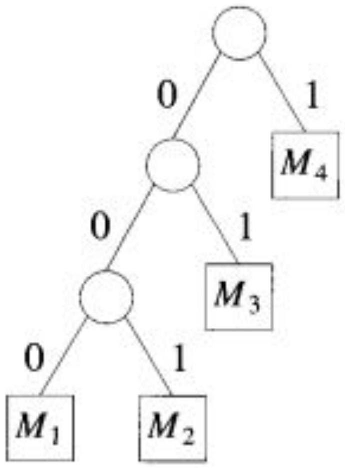
```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

Algorithm Tree(n)
// list is a global list of n single node
// binary trees as described above.
{
    for i := 1 to n − 1 do
    {
        pt := new treenode; // 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 list is the merge tree.
}
```

# Example



# Huffman Codes

- To obtain optimal set of codes for messages $M_1, M_2, \ldots, M_n$
- Generate two way merge tree, with external nodes representing messages
- Label left branch with 0 and right branch with 1



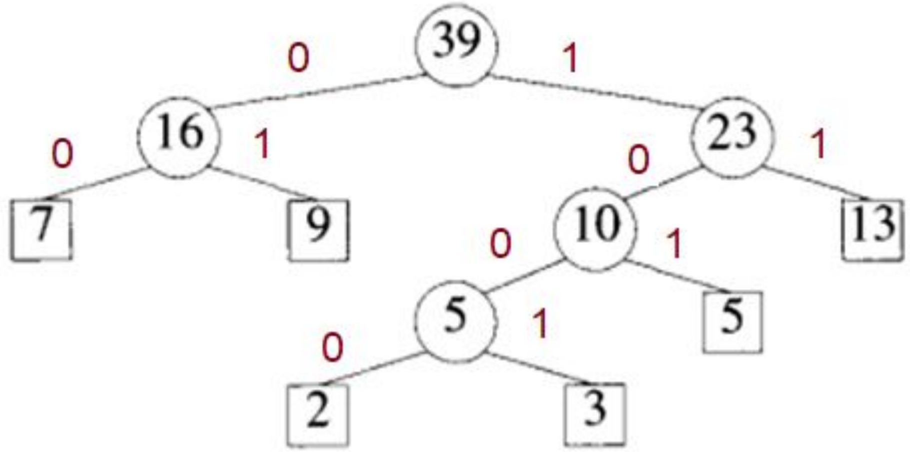

| Message | Code |
|---|---|
| $M_1$ | 000 |
| $M_2$ | 001 |
| $M_3$ | 01 |
| $M_4$ | 1 |

Decode Message 0110010001

0110010001 $M_3M_4M_2M_1M_4$

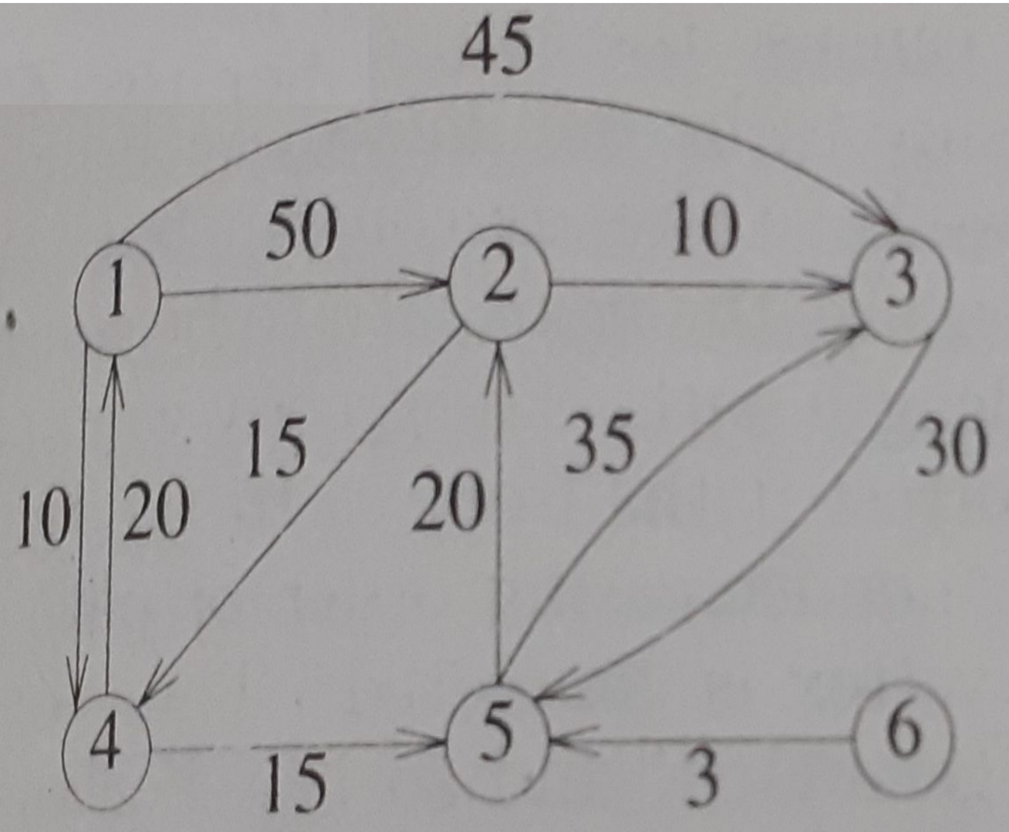
# Huffman codes

- Messages $M_1 \ldots M_7$ with frequencies {2, 3, 5, 7, 9, 13}




| Message | Code |
|---|---|
| $M_1$ | 1000 |
| $M_2$ | 1001 |
| $M_3$ | 101 |
| $M_4$ | 00 |
| $M_5$ | 01 |
| $M_6$ | 11 |

# Single Source Shortest Path




(a) Graph

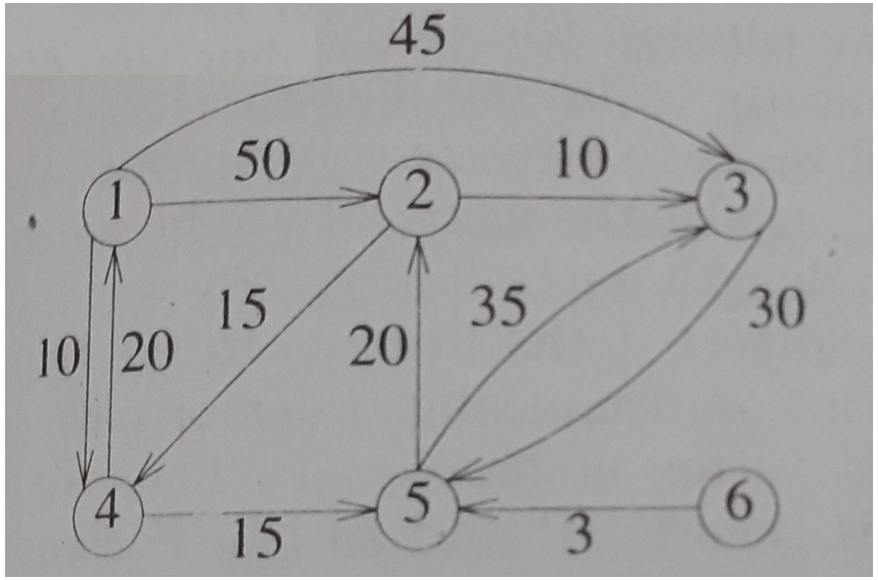
| | *Path* | *Length* |
|---|---|---|
| 1) | 1, 4 | 10 |
| 2) | 1, 4, 5 | 25 |
| 3) | 1, 4, 5, 2 | 45 |
| 4) | 1, 3 | 45 |

(b) Shortest paths from 1

# Single Source Shortest path algorithm

- v – Starting vertex
- Dist(j) – length of the shortest path from vertex v to vertex j
- S – set of vertices to which shortest paths have already been generated

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| S | 0 | 0 | 0 | 0 | 0 | 0 |
| Dist | 0 | 50 | 45 | 10 | ∞ | ∞ |



**Step 1 : Initial : Starting vertex 0**

Set S[1] = 1

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ |
|---|---|---|---|---|---|---|
| S | 1 | 0 | 0 | 0 | 0 | 0 |
| Dist | 0 | 50 | 45 | 10 | ∞ | ∞ |
| Path | | 1-2 | 1-3 | 1-4 | | |

**Step 2 : Minimum Dist vertex $v_4$**

Set S[4] = 1

$Dist(v_2) = \min(Dist(v_2), Dist(v_4) + c(v_4, v_2))$
$= \min(50, 10+inf) = 50$

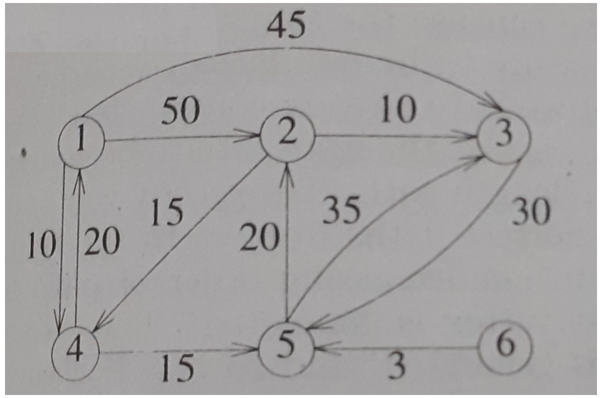
$Dist(v_3) = \min(Dist(v_3), Dist(v_4) + c(v_4, v_3))$
$= \min(45, 10+inf) = 45$

$Dist(v_5) = \min(Dist(v_2), Dist(v_4) + c(v_4, v_2))$
$= \min(inf, 10+15) = 25$

$Dist(v_6) = \min(Dist(v_6), Dist(v_4) + c(v_4, v_6))$
$= \min(inf, 10+inf) = inf$

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ |
|---|---|---|---|---|---|---|
| S | 1 | 0 | 0 | 1 | 0 | 0 |
| Dist | 0 | 50 | 45 | 10 | 25 | ∞ |
| Path | | 1-2 | 1-3 | 1-4 | 1-4-5 | |

$$DIST(w) \leftarrow \min(DIST(w), DIST(u) + COST(u, w))$$




**Step 3 : Minimum Dist vertex $v_5$**

Set S[5] = 1

$Dist(v_2) = \min(Dist(v_2), Dist(v_5) + c(v_5, v_2))$
$= \min(50, 25+20) = 45$

$Dist(v_3) = \min(Dist(v_3), Dist(v_5) + c(v_5, v_3))$
$= \min(45, 25+inf) = 45$

$Dist(v_6) = \min(Dist(v_6), Dist(v_5) + c(v_5, v_6))$
$= \min(inf, 25+inf) = inf$

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ |
|---|---|---|---|---|---|---|
| S | 1 | 0 | 0 | 1 | 1 | 0 |
| Dist | 0 | 45 | 45 | 10 | 25 | ∞ |
| Path | | 1-4-5<br>-2 | 1-3 | 1-4 | 1-4-5 | |

# Greedy Algorithm to generate Shortest paths

```
Algorithm ShortestPaths(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 a 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
    { // Initialize S.
        S[i] := false; dist[i] := cost[v, i];
    }
    S[v] := true; dist[v] := 0.0; // Put v in S.
    for num := 2 to n 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
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                dist[w] := dist[u] + cost[u, w];
    }
}
```