Design and Analysis of Algorithms
EID305

Module II: The Greedy Method

Prof. PV Nageswara Rao
Dept. of CSE

Greedy Method

General Method:

Given a problem with *n* inputs, we are required to obtain a subset that maximizes or minimizes a given objective function subject to some constraints.

Feasible solution — any subset that satisfies constraints

Optimal solution — a feasible solution that maximizes or minimizes the objective function

- The Greedy Algorithm works in stages, considering one input at a time.
- At each stage a decision is made whether to include the considered item or not.
- Therefore, the input data should be in a particular order.
- Subset Paradigm
  - Knapsack Problem
  - Job Sequencing with deadlines
  - Minimum Cost Spanning Tree
- Ordering Paradigm
  - Optimal storage on tapes
  - Optimal Merge Patterns
  - Single Source Shortest paths

procedure Greedy *(A, n)*

```
begin
    solution ← Ø;
    for i ← 1 to n do
        x ← Select (A);   // based on the objective
                          //  function
        if Feasible (solution, x),
            then solution ← Union (solution, x);
end;
```

Select: A greedy procedure, based on a given objective function, which selects input from *A*, removes it and assigns its value to *x*.

Feasible: A Boolean function to decide if *x* can be included into solution vector (without violating any given constraint).

Union: Combines x with the solution

Knapsack Problem

The problem:

Given a knapsack with a certain capacity M, n objects, are to be put into the knapsack, each has a weight $w_1, w_2, \cdots, w_n$ and a profit $p_1, p_2, \cdots, p_n$ if put in the knapsack

<u>The goal</u> is find $(x_1, x_2, \cdots, x_n)$ where $0 \le x_i \le 1$

s.t. $\sum_{i=1}^{n} p_i x_i$ is maximized and $\sum_{i=1}^{n} w_i x_i \le M$

<u>Note:</u> A part/portion of an object can be selected.
or x_i can be any fraction between 0 and 1.

The knapsack problem

- n objects, each with a weight $w_i > 0$

  a profit $p_i > 0$

  capacity of knapsack: M

Maximize $\sum_{1 \le i \le n} p_i x_i$

Subject to $\sum_{1 \le i \le n} w_i x_i \le M$

$0 \le x_i \le 1, \ 1 \le i \le n$

Example:

$$n = 3$$

$$M = 20$$

$$(w_1, w_2, w_3) = (18,15,10)$$

$$(p_1, p_2, p_3) = (25,24,15)$$

Greedy Strategy#1: Profits are ordered in nonincreasing order (1,2,3)

$$(x_1, x_2, x_3) = (1, \frac{2}{15}, 0)$$

$$\sum_{i=1}^{3} p_i x_i = 25 \times 1 + 24 \times \frac{2}{15} + 15 \times 0 = 28.2$$

Greedy Strategy#2: Weights are ordered in nondecreasing order (3,2,1)

$$(x_1, x_2, x_3) = (0, \frac{2}{3}, 1)$$

$$\sum_{i=1}^{3} p_i x_i = 25 \times 0 + 24 \times \frac{2}{3} + 15 \times 1 = 31$$

Greedy Strategy#3: *p/w* are ordered in nonincreasing order (2,3,1)

$$\left.\begin{aligned} \frac{p_1}{w_1} &= \frac{25}{18} = 1.4 \\ \frac{p_2}{w_2} &= \frac{24}{15} = 1.6 \\ \frac{p_3}{w_3} &= \frac{15}{10} = 1.5 \end{aligned}\right\} \Rightarrow (2,3,1)$$

Optimal solution

$$(x_1, x_2, x_3) = (0,\ 1,\ \frac{1}{2})$$

$$\sum_{i=1}^{3} p_i x_i = 25 \times 0 + 24 \times 1 + 15 \times \frac{1}{2} = \underline{31.5}$$

The knapsack algorithm

- The greedy algorithm:

  Step 1: Sort p_i/w_i into nonincreasing order.

  Step 2: Put the objects into the knapsack according to the sorted sequence as possible as we can.

- e. g.

  $n = 3, M = 20, (p_1, p_2, p_3) = (25, 24, 15)$

  $(w_1, w_2, w_3) = (18, 15, 10)$

  Sol: $p_1/w_1 = 25/18 = 1.39$

  $p_2/w_2 = 24/15 = 1.6$

  $p_3/w_3 = 15/10 = 1.5$

  Optimal solution: $x_1 = 0, x_2 = 1, x_3 = 1/2$

  total profit = 24 + 7.5 = 31.5

Algorithm greedy knapsack(m, n)

```
//p[1:n], w[1:n] contains 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 size and x[1:n] is the solution vector.
{   for i:=1 to n do
          x[i]=0.0;
      u:=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];
}
```

Analysis:

Sort the *p/w*, such that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$

Show that the ordering is the best.

<u>Proof by contradiction:</u>

Given some knapsack instance

Suppose the objects are ordered s.t. $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$

let the greedy solution be $X = (x_1, x_2 \cdots, x_n)$

Show that this ordering is optimal

Case1: $X = (1,1,\cdots,1)$ it's optimal

Case2: $X = (1,1,\cdots,x_j,0,\cdots,0)$ s.t. $\sum_{i=1}^{n} w_i x_i = M$

where $0 \leq x_j \leq 1$

Assume X is not optimal, and then there exists $Y = (y_1, y_2, \cdots, y_n)$

$$\text{s.t.} \sum_{i=1}^{n} p_i y_i > \sum_{i=1}^{n} p_i x_i \quad \text{and } Y \text{ is optimal}$$

examine X and Y, let y_k be the 1st one in Y that $y_k \neq x_k$.

$$X = (\underbrace{x_1, x_2, \cdots\cdots, x_{k-1}}_{\text{same}}, x_k, \cdots\cdots, x_n)$$

$$y_k \neq x_k \Rightarrow y_k < x_k$$

$$Y = (\overbrace{y_1, y_2, \cdots\cdots, y_{k-1}}, y_k, \cdots\cdots, y_n)$$

Now we increase y_k to x_k and decrease as many of $(y_{k+1}, \cdots, y_n)$ as necessary, so that the capacity is still M.

Let this new solution be $Z = (z_1, z_2, \cdots, z_n)$

where $z_i = x_i \quad \forall 1 \le i \le k$

$$and\,(z_k - y_k)\cdot w_k = \sum_{i=k+1}^{n} (y_i - z_i)\cdot w_i$$

$$\sum_{i=1}^{n} p_i z_i = \sum_{i=1}^{n} p_i y_i + (z_k - y_k)\cdot p_k - \sum_{i=k+1}^{n} (y_i - z_i)\cdot p_i$$

$$\because \frac{p_k}{w_k} \ge \frac{p_i}{w_i} \qquad \forall i \ge k+1$$

$$\therefore p_i \le \left(\frac{p_k}{w_k}\right)\cdot w_i$$

$$\sum_{i=1}^{n} p_i z_i \ge \sum_{i=1}^{n} p_i y_i + \left[(z_k - y_k)\cdot w_k - \sum_{i=k+1}^{n} (y_i - z_i)\cdot w_i\right]\cdot \frac{p_k}{w_k} = \sum_{i=1}^{n} p_i y_i$$

So,

if $\quad profit(z) > profit(y) \qquad (\rightarrow\leftarrow)$

else $\quad profit(z) = profit(y)$

(Repeat the same process.
At the end, Y can be transformed into X.
$\Rightarrow X$ is also optimal.
Contradiction! $\rightarrow\leftarrow$)

Some web resources:
https://www.radford.edu/~nokie/classes/360/greedy.html

Q & A

1. **Minimum Spanning Tree** (For Undirected Graph)

The problem:

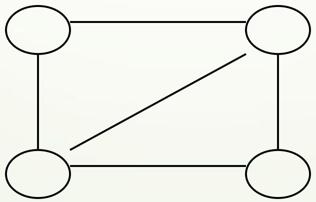
1) Tree

   A *Tree* is connected graph with no cycles.

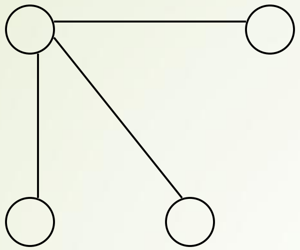
2) Spanning Tree

   A *Spanning Tree* of *G* is a tree which contains all vertices in *G*.

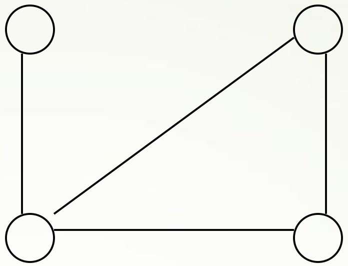
   Example: *G:*



b) Is *G* a Spanning Tree?



Key: Yes



Key: No

Note: Connected graph with *n* vertices and exactly $n - 1$ edges is Spanning Tree.

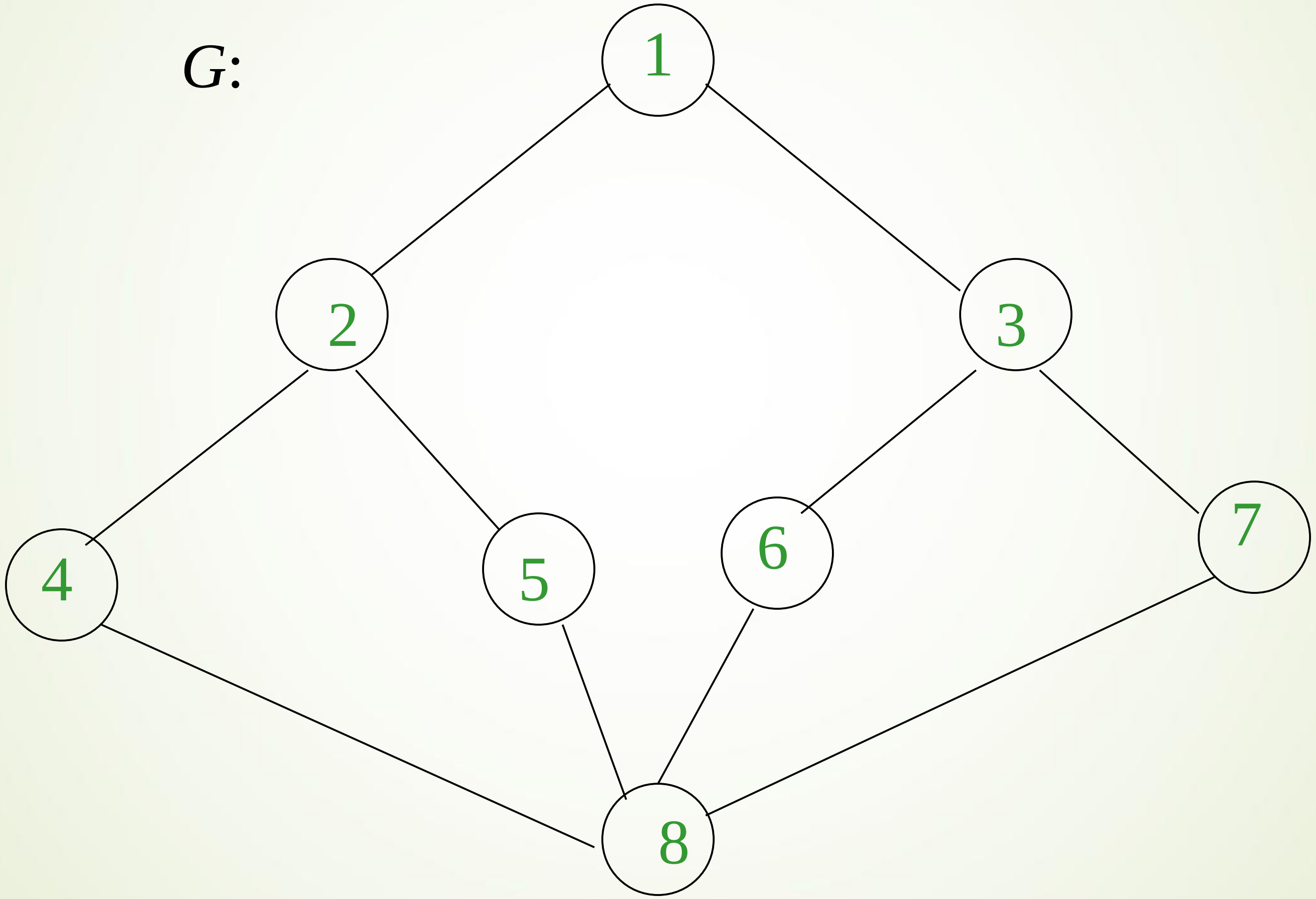
3) Minimum Spanning Tree

Assign weight to each edge of *G*, then *Minimum Spanning Tree* is the Spanning Tree with minimum total weight.

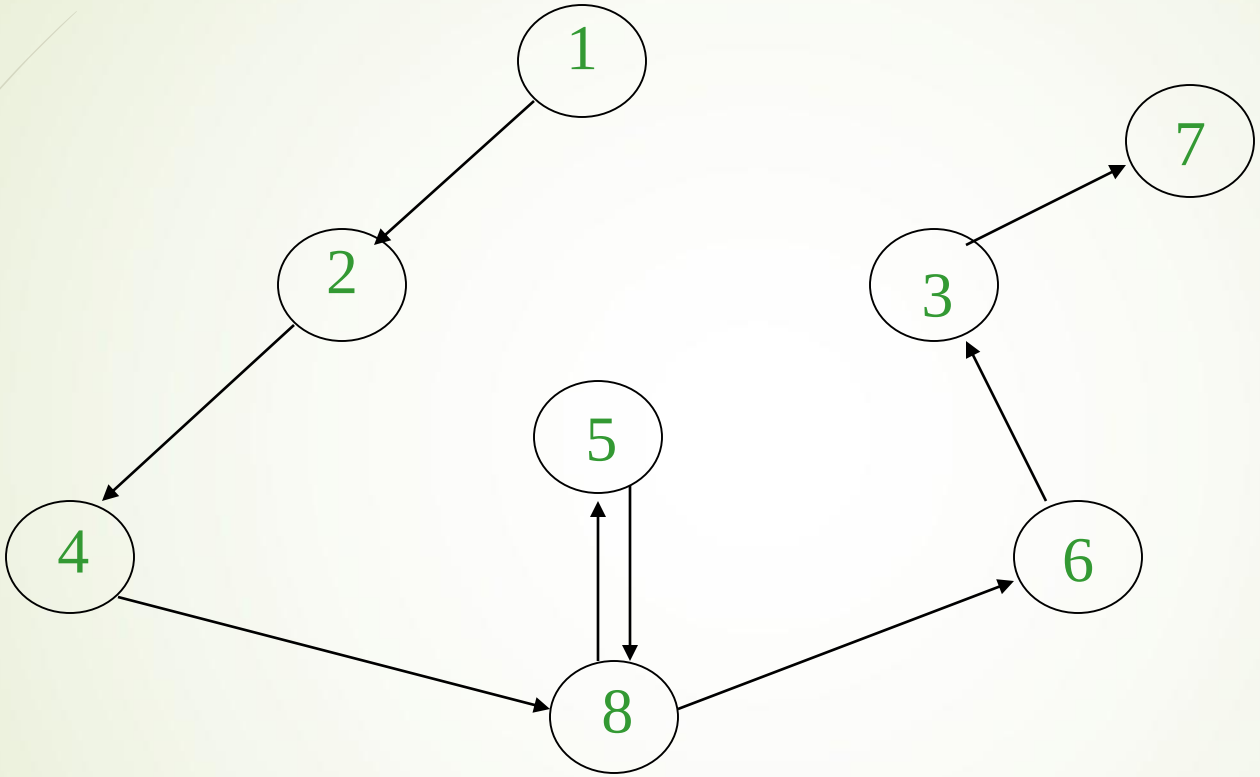
Example:

a) Edges have the same weight

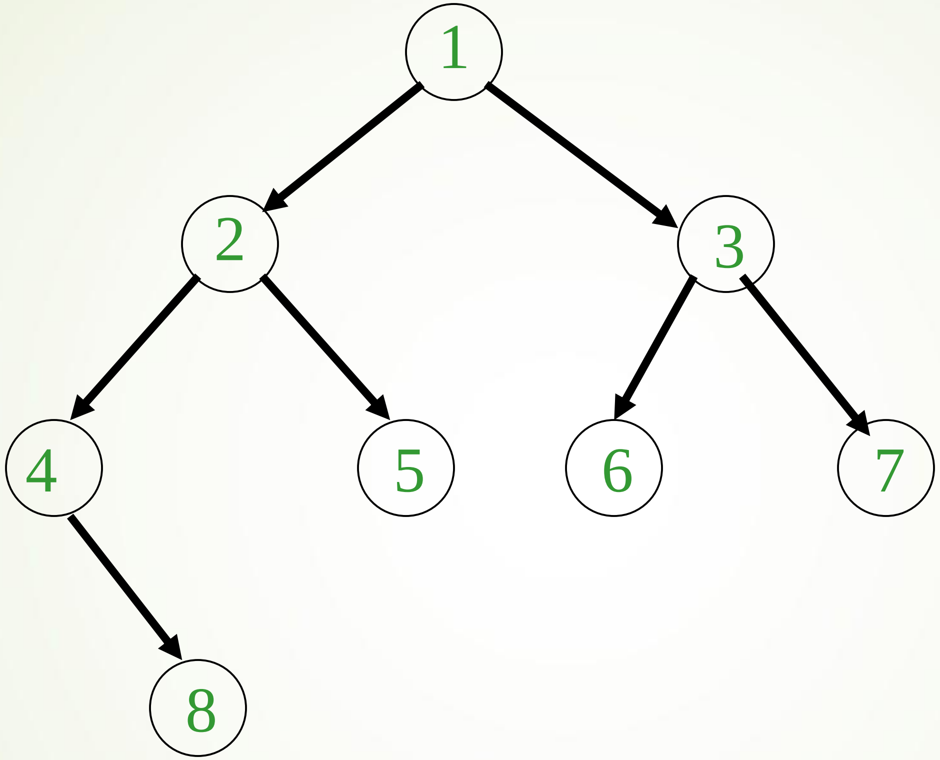
G:



- DFS (Depth First Search)

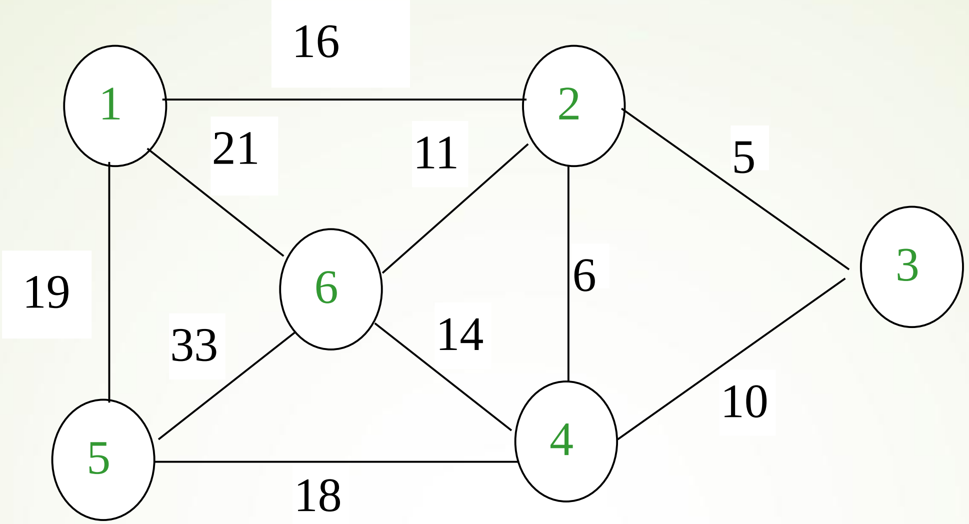


- BFS (Breadth First Search)

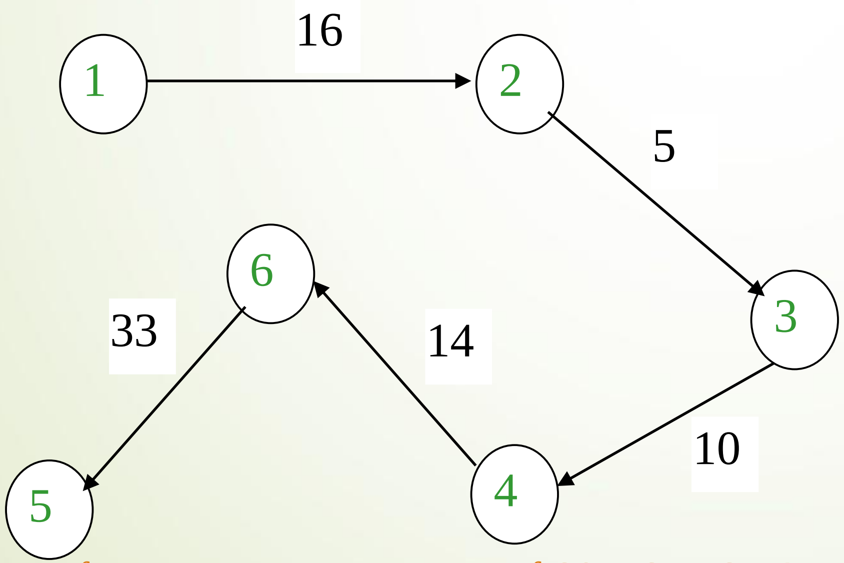


b) *Edges have different weights*

<u>*G:*</u>

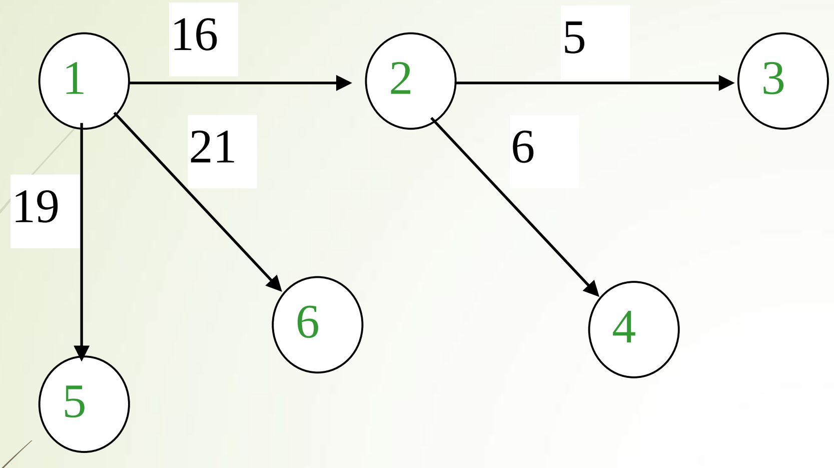


- DFS



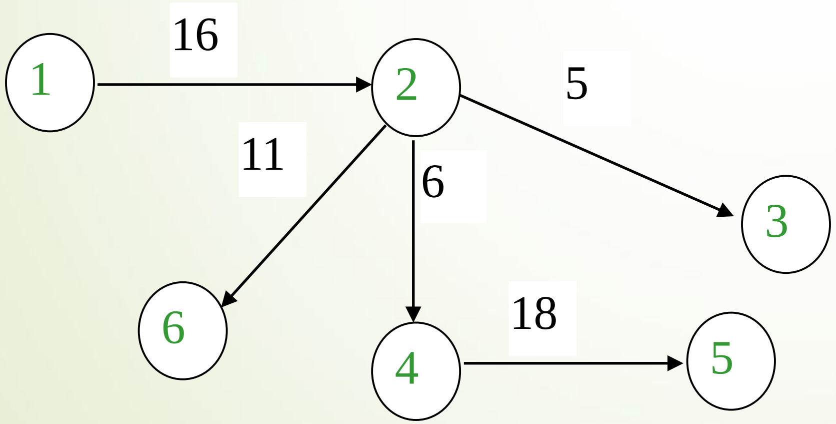
$$Cost = 16 + 5 + 10 + 14 + 33$$
$$= 78$$

- BFS



$$Cost = 16 + 19 + 21 + 5 + 6$$
$$= 67$$

- Minimum Spanning Tree (with the least total weight)



$$Cost = 16 + 5 + 6 + 11 + 18$$
$$= 56$$

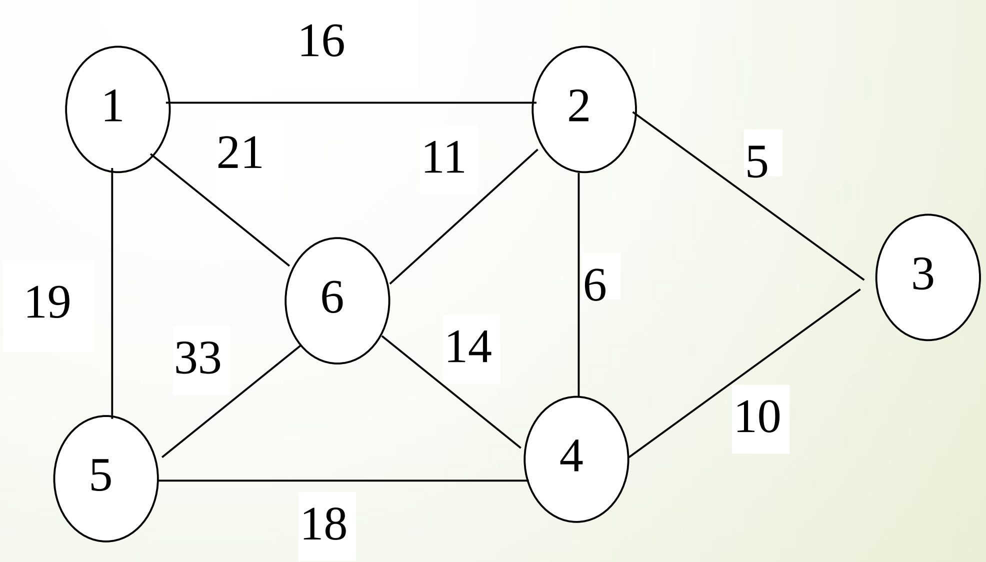
Algorithms:

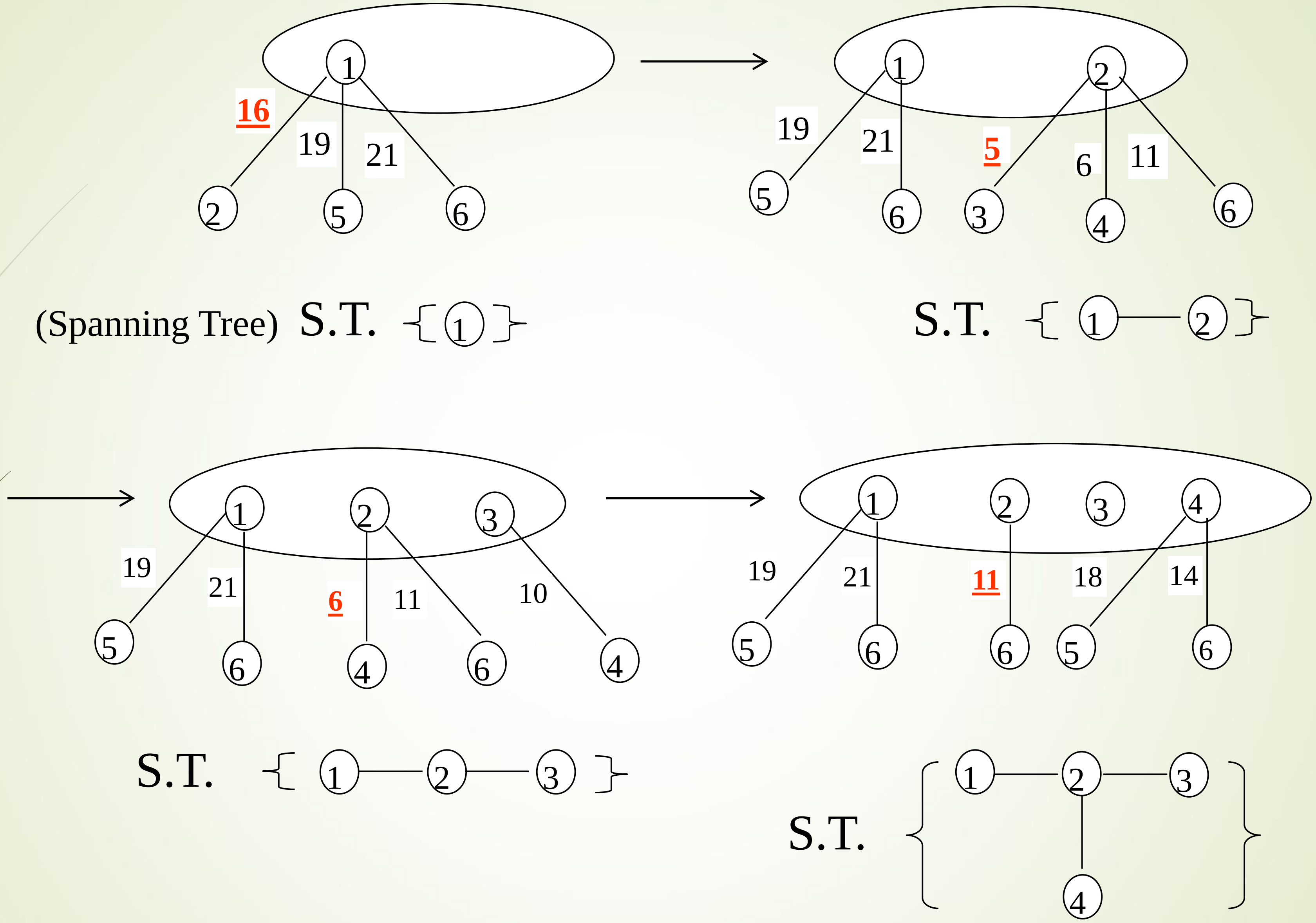
1) *Prim's Algorithm* *(Minimum Spanning Tree)*

Basic idea:

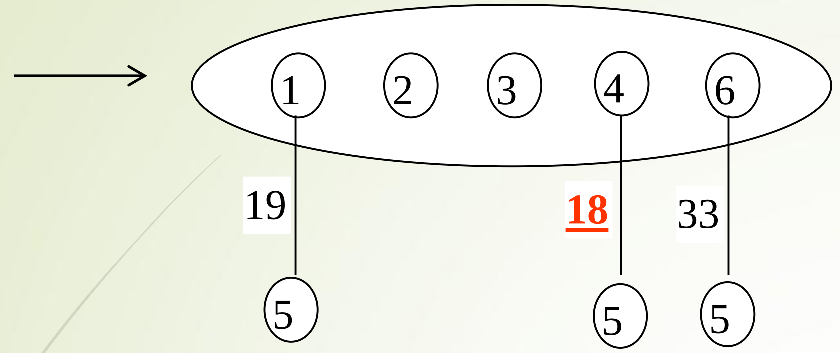
Start from vertex 1 and let T ← Ø (T will contain all edges in the S.T.); the next edge to be included in T is the minimum cost edge(u, v), s.t. u is in the tree and v is not.

Example: *G*

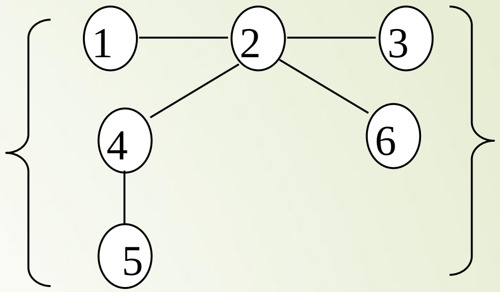




16
19
21
(Spanning Tree) S.T.
S.T.
19
21
5
6
11
19
21
6
11
10
S.T.
19
21
11
18
14
S.T.



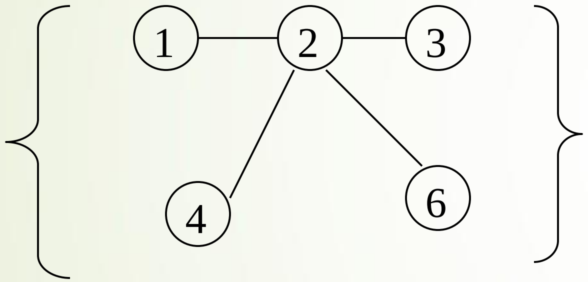
S.T.



Cost = 16+5+6+11+18=56

Minimum Spanning Tree

S.T.



(*n* – # of vertices, *e – # of edges)*

It takes $O(n)$ steps. Each step takes $O(e)$ *and* $e \leq n(n\text{-}1)/2 \Rightarrow O(n^2)$.

Therefore, it takes $O(n^3)$ *time.*

With clever data structure, it can be implemented in $O(n^2)$.

Prim's Algorithm

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 infinity if no edge (i,j) exist.

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.

Prim’s Algorithm

```
Algorithm Prim(E, cost, n, t)
{
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
    {
        If cost[i,l] <cost[i,k]) then near[i]=l;
        Else  near[i]=k;
    }
near[k]=near[l]=0;
```

Prim's Algorithm Contd..

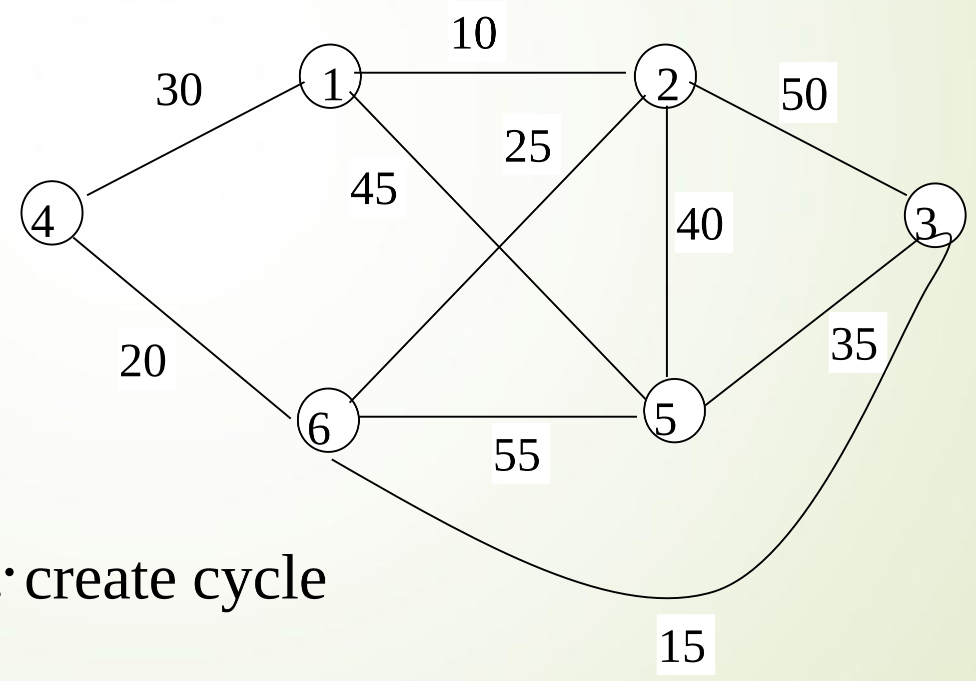

```
For(i=2 to n-1) do
 {
 //Find n-2 additional edges for t.
     Let j be an index such that near[j] ≠0 and
     cost[j,near[j]] is minimum;
      t[i,1]=j;
      t[i,2]=near[j];
     mincost = mincost + cost[j,near[j]];
     near[j]=0;
     for k= 1 to n do
        if((near[k] ≠ 0) and
                 (cost[k,near[k]] >cost[k,j]))then
             near[k] =j;
     }
return mincost; }
```

2) Kruskal's Algorithm

Basic idea:

Don't care if *T* is a tree or not in the intermediate stage, as long as the including of a new edge will not create a cycle, we include the minimum cost edge

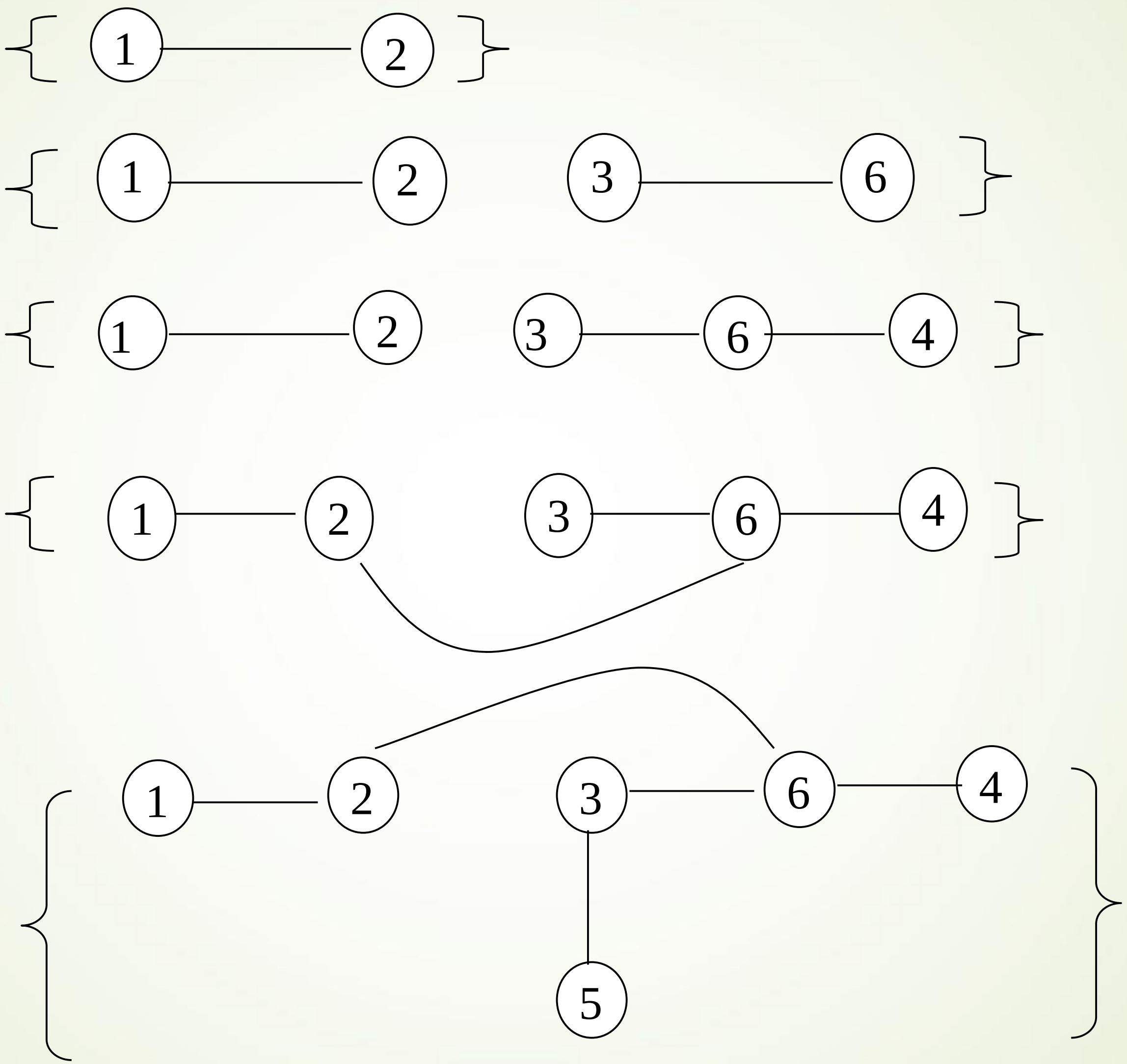
Example:



Step 1: Sort all of edges

(1,2)	10	√	
(3,6)	15	√	
(4,6)	20	√	
(2,6)	25	√	
(1,4)	30	×	reject ∵ create cycle
(3,5)	35	√	

Step 2: T

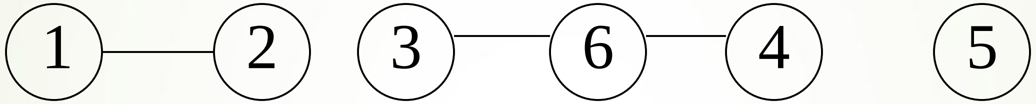


<u>How to check</u>:

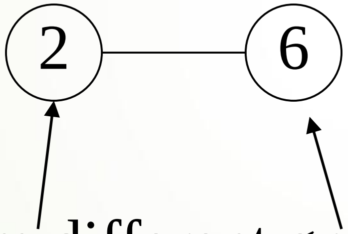
adding an edge will <u>create a cycle or not</u>?

If Maintain a <u>set</u> for each group
(initially each node represents a set)

Ex: <u>set1</u> <u>set2</u> <u>set3</u>



∴ new edge



from different groups ⇒ no cycle created

Data structure to store sets so that:

i. The group number can be easily found, and

ii. Two sets can be easily merged

Kruskal's algorithm

While (T contains fewer than n-1 edges) and ($E \neq \emptyset$) do
Begin

    Choose an edge (v,w) from E of lowest cost;
    Delete (v,w) from E;
    If (v,w) does not create a cycle in T
    then add (v,w) to T
    else discard (v,w);

End;

With clever data structure, it can be implemented in *O(e log e)*.

Kruskals Algorithm

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.

```
Algorithm Kruskal(E,cost,n,t)
{
        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
```

Kruskals Algorithm(contd..)


```
while((i<=n-1) and (heap not empty)) do
{
Delete a minimum cost edge (u,v) from the heap and
reheapify 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);
      }
}
```

Kruskals Algorithm(contd..)

```
if (i ≠ n-1) then
    print("no spanning tree");
else
    return mincost;
}
```

So, complexity of Kruskal is $O(eLoge)$

$$\because e \le \frac{n(n-1)}{2} \Rightarrow Loge \le Logn^2 = 2Logn$$

$$\Rightarrow O(eLoge) = O(eLogn)$$

3) Comparing Prim's Algorithm with Kruskal's Algorithm

i. Prim's complexity is $O(n^2)$

ii. Kruskal's complexity is $O(eLogn)$

if G is a complete (dense) graph,
Kruskal's complexity is $O(n^2 Logn)$
if G is a sparse graph,
Kruskal's complexity is $O(nLogn)$.

2. Dijkstra's Algorithm for Single-Source Shortest Paths

The problem: Given directed graph $G = (V, E)$,
a weight for each edge in G,
a source node v_0,

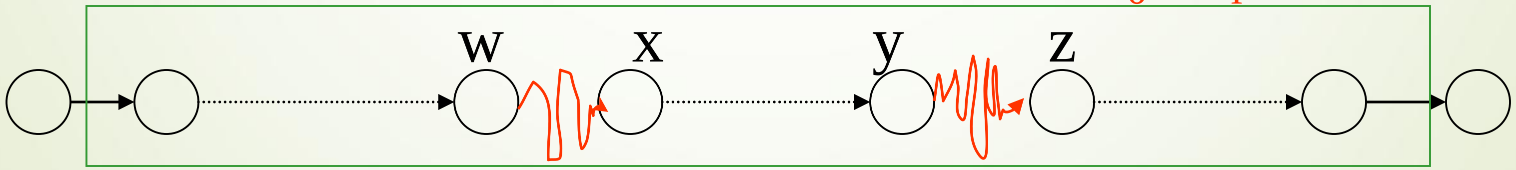
Goal: determine the (length of) shortest paths from v_0 to all the remaining vertices in G

Def: Length of the path: Sum of the weight of the edges

Observation:

May have more than 1 paths between w and x (y and z)
But each individual path must be minimal length

(in order to form an overall shortest path form v_0 to v_i)



V_0 shortest paths from v_0 to v_i V_i

Notation

cost adjacency matrix $Cost$, $\forall 1 \le a,b \le |V|$

$$Cost\ (a,\ b) = \begin{cases} \text{cost from vertex i to vertex j} & \text{if there is a edge} \\ 0 & \text{if } a = b \\ \infty & \text{otherwise} \end{cases}$$

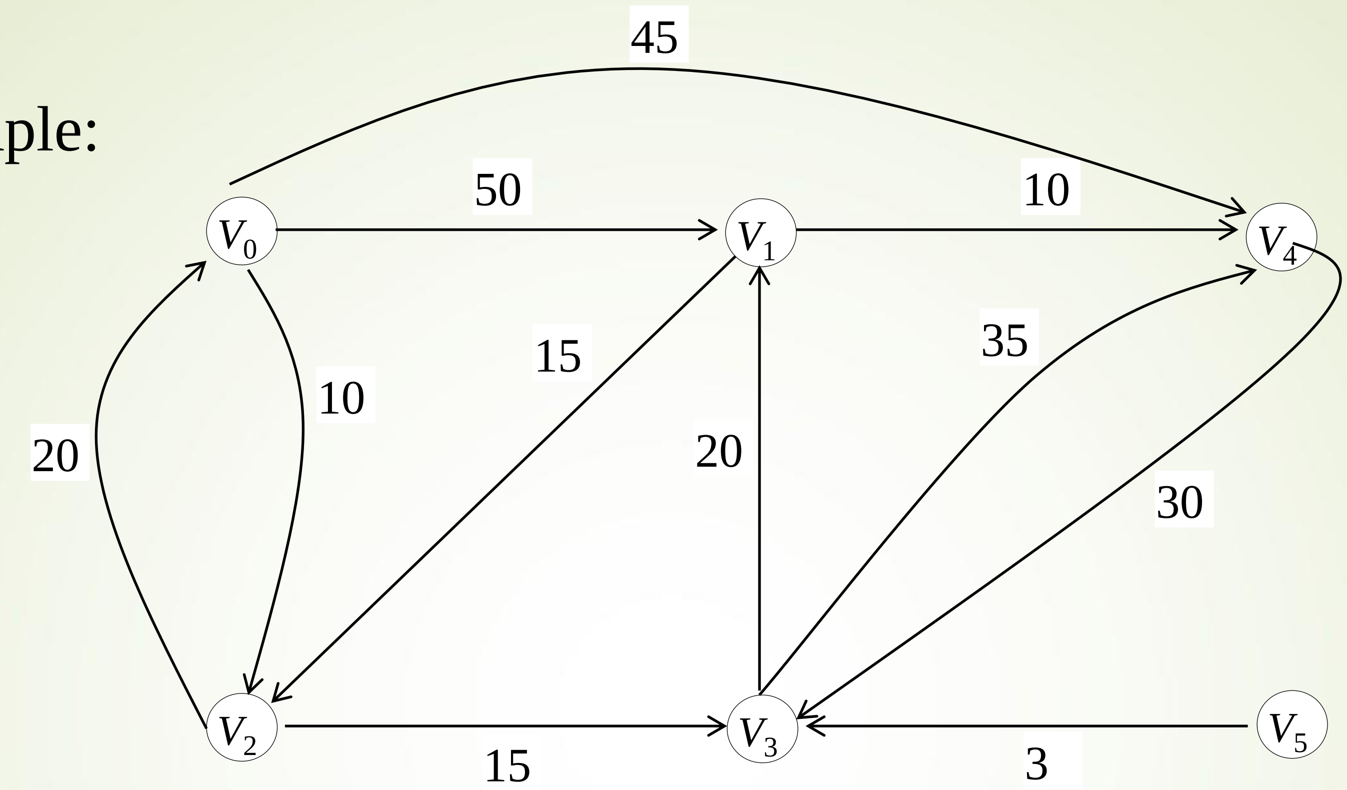
$$s(w) = \begin{cases} 1 & \text{if shortest path } (v_0\,,\ w) \text{ is defined} \\ 0 & \text{otherwise} \end{cases}$$

$Dist(j)$ $\forall j$ in the vertex set V

= the length of the shortest path from v_0 to j

$From(j) = i$ if i is the predecessor of j along the shortest path from v_0 to j

Example:

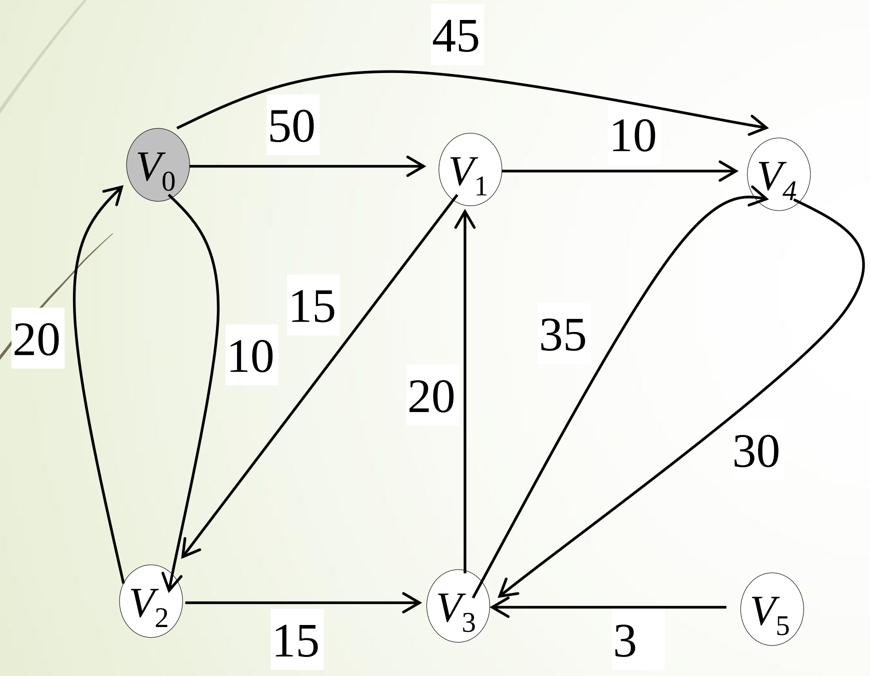


a) Cost adjacent matrix

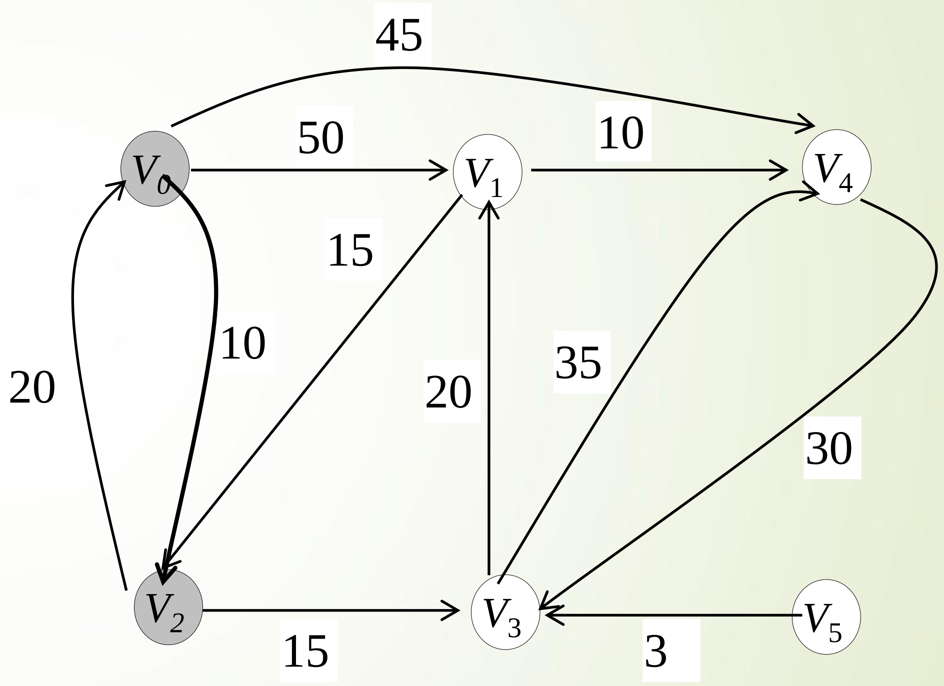
$$\begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccc} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 10 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 20 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix} \end{array}$$

b) Steps in Dijkstra's Algorithm

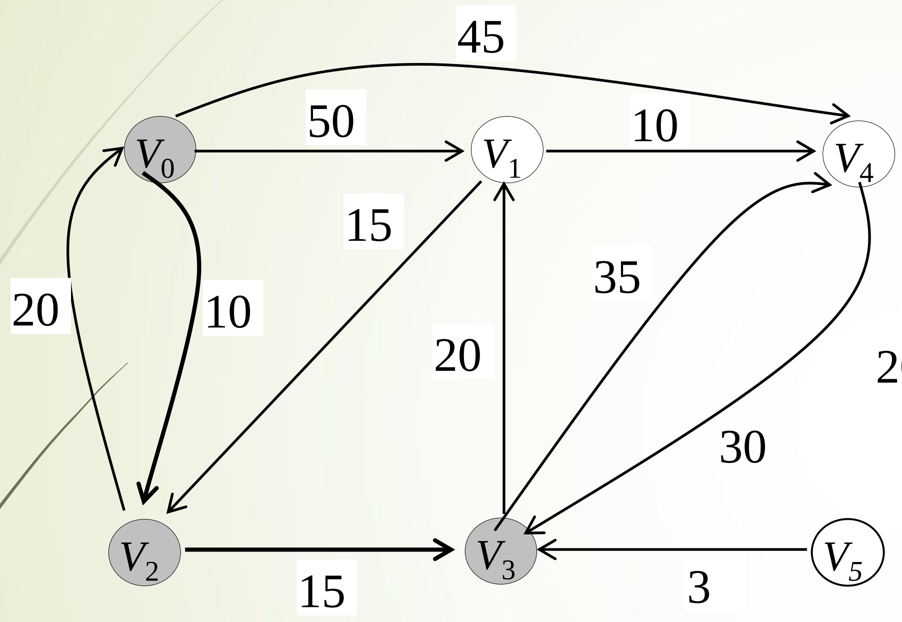
1. *Dist* $(v_0) = 0$, *From* $(v_0) = v_0$

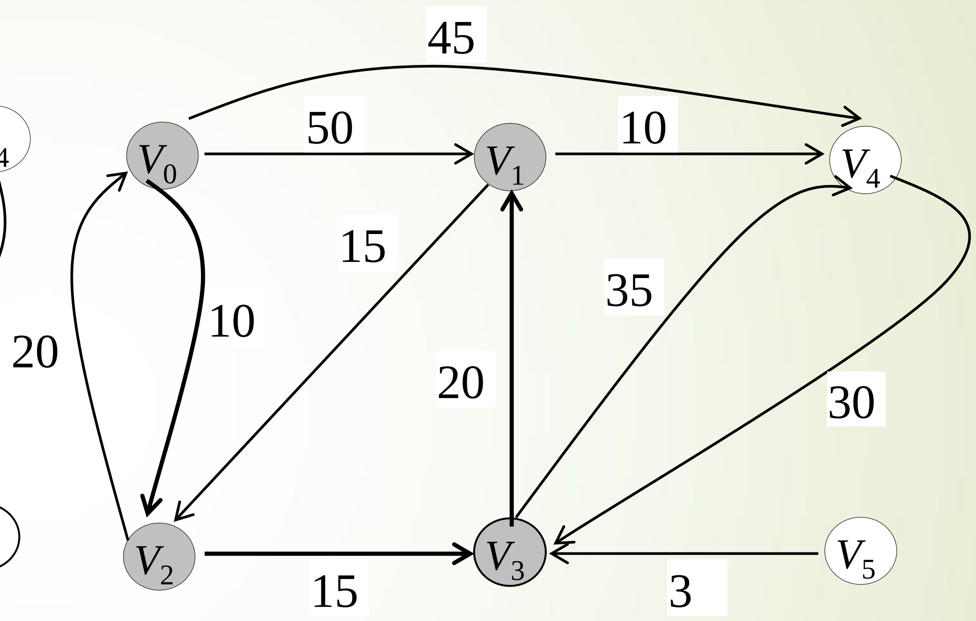


2. *Dist* $(v_2) = 10$, *From* $(v_2) = v_0$

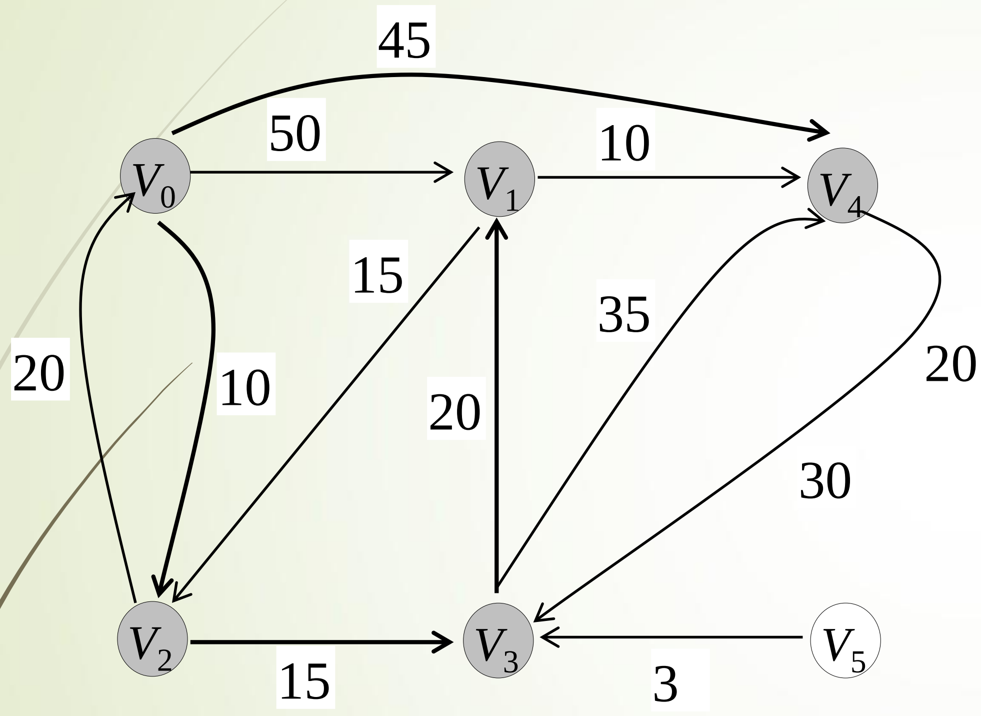


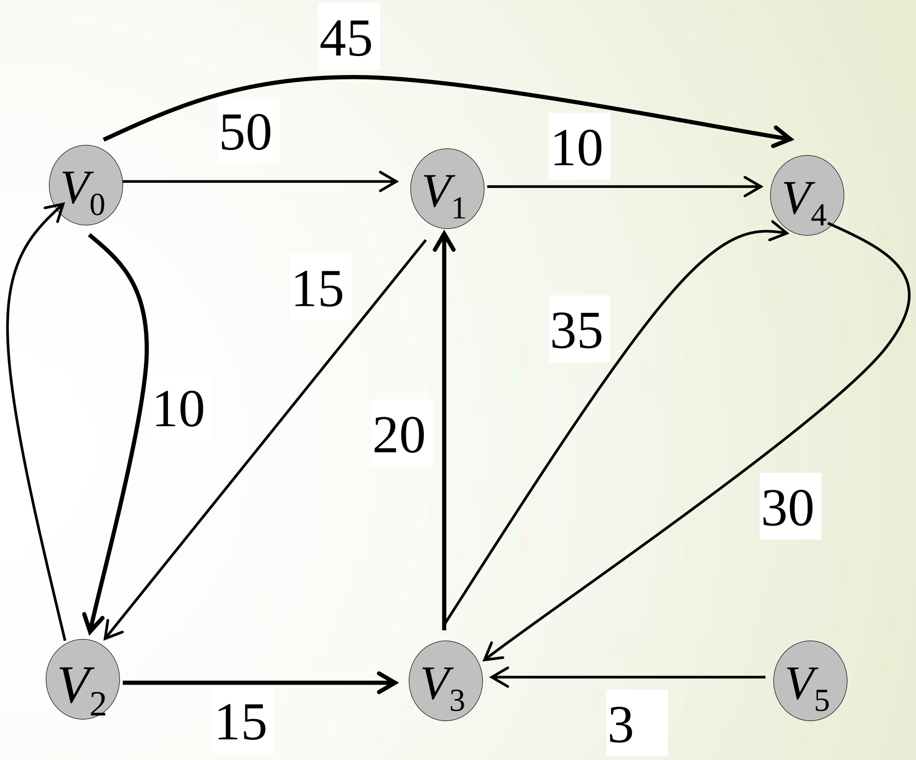
3. *Dist* $(v_3) = 25$, *From* $(v_3) = v_2$ 4. *Dist* $(v_1) = 45$, *From* $(v_1) = v_3$





$5.\ Dist\ (v_4) = 45,\ From\ (v_4) = v_0$ $\quad$ $6.\ Dist\ (5) = \infty$





c) Shortest paths from source v_0

$v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1$	45
$v_0 \rightarrow v_2$	10
$v_0 \rightarrow v_2 \rightarrow v_3$	25
$v_0 \rightarrow v_4$	45
$v_0 \rightarrow v_5$	∞

Dijkstra's algorithm:

procedure Dijkstra (*Cost*, *n*, *v*, *Dist*, *From*)
// *Cost*, *n*, *v* are input, *Dist*, *From* are output

begin
for $i \leftarrow 1$ to n do

$s(i) \leftarrow 0;$

$Dist(i) \leftarrow Cost(v, i);$

$From(i) \leftarrow v;$

$s(v) \leftarrow 1;$

for $num \leftarrow 1$ to $(n-1)$ do

choose u s.t. $s(u) = 0$ and $Dist(u)$ is minimum;

$s(u) \leftarrow 1;$

for all w with $s(w) = 0$ do

if $(Dist(u) + Cost(u, w) < Dist(w))$

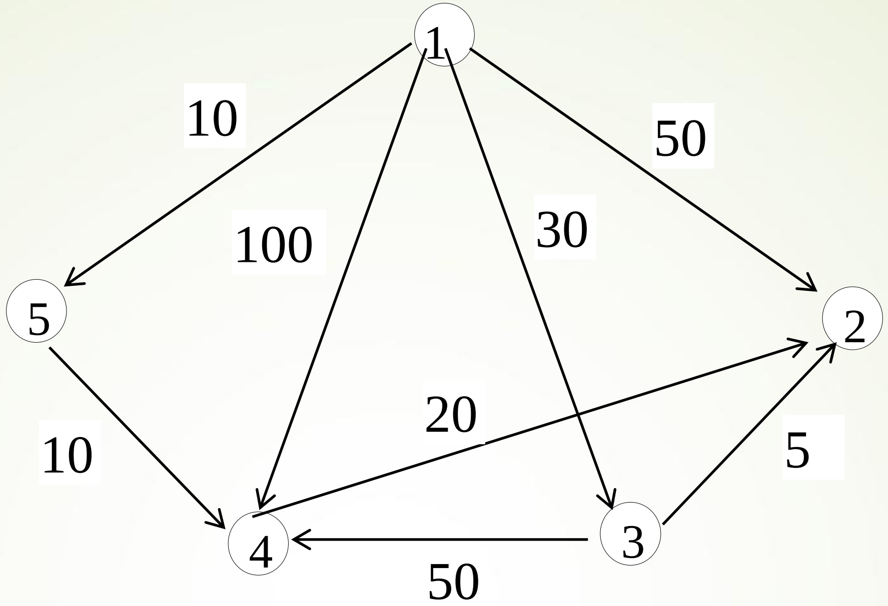
$Dist(w) \leftarrow Dist(u) + Cost(u, w);$

$From(w) \leftarrow u;$

end;

(cont. next page)

Ex:

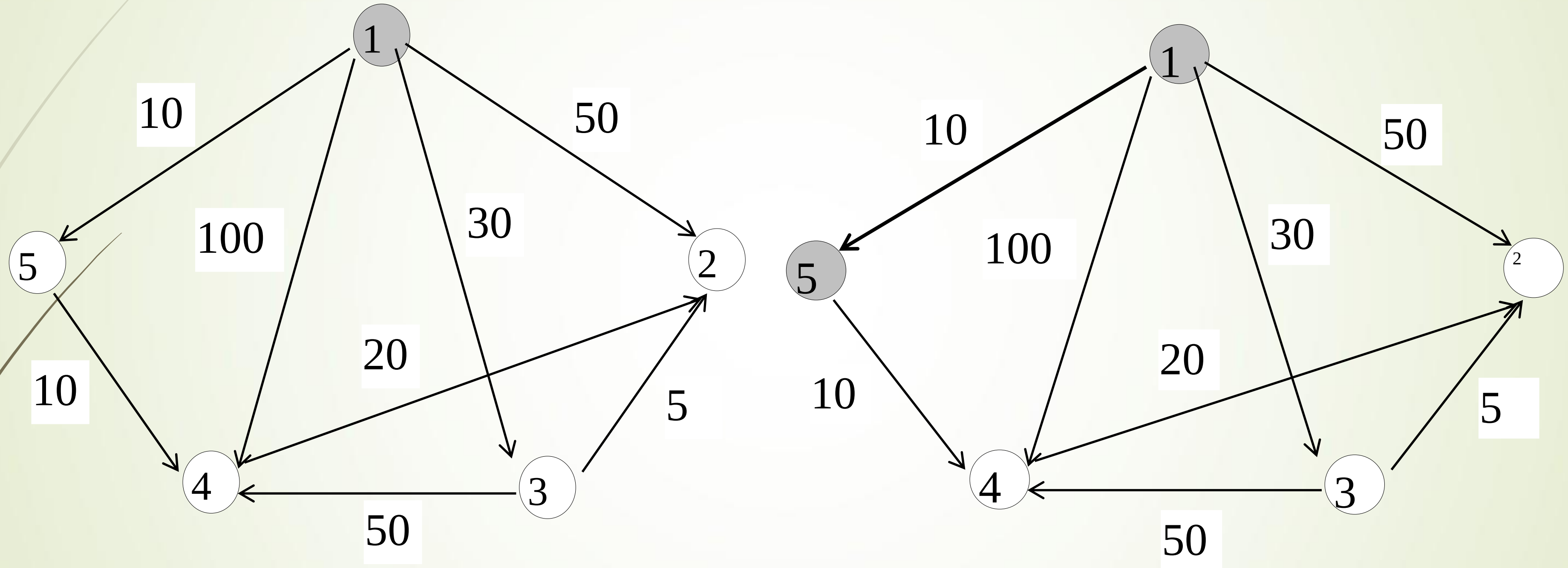


a) Cost adjacent matrix

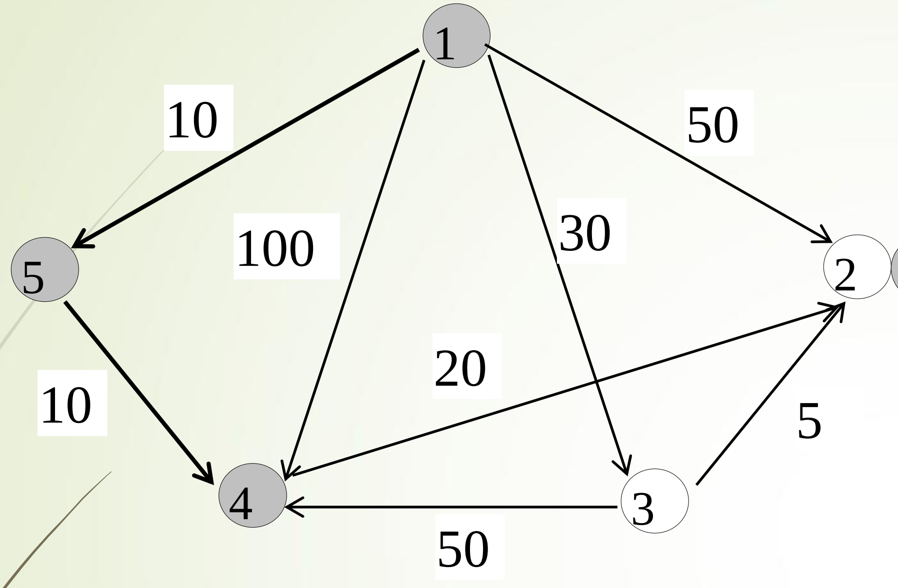
$$
\begin{array}{c c}
 & \begin{array}{ccccc} 01 & 02 & 03 & 04 & 05 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{bmatrix}
0 & 50 & 30 & 100 & 10 \\
\infty & 0 & \infty & \infty & \infty \\
\infty & 5 & 0 & 50 & \infty \\
\infty & 20 & \infty & 0 & \infty \\
\infty & \infty & \infty & 10 & 0
\end{bmatrix}
\end{array}
$$

b) Steps in Dijkstra's algorithm

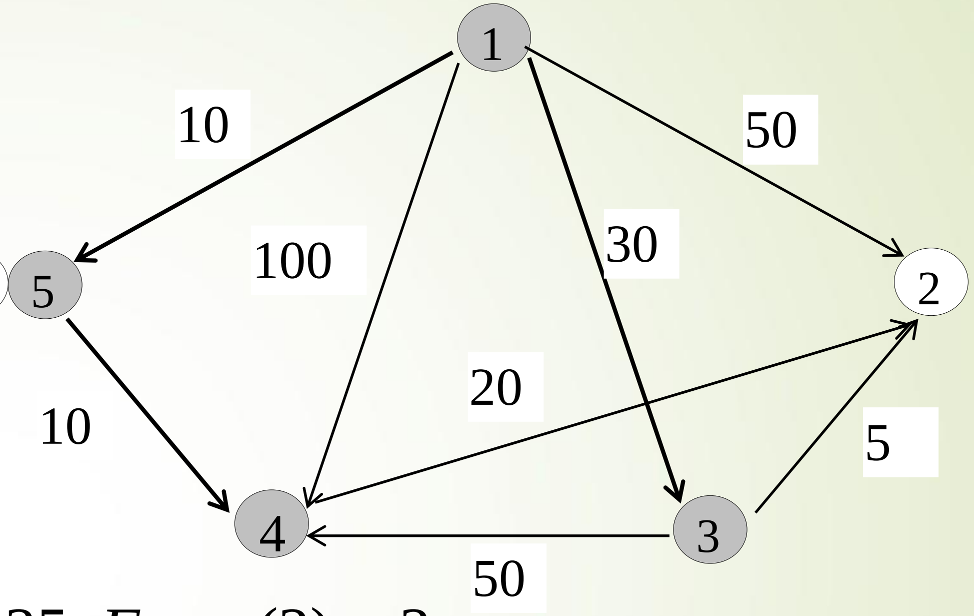
1. *Dist* (1) = 0, *From* (1) = 1 2. *Dist* (5) = 10, *From* (5) = 1



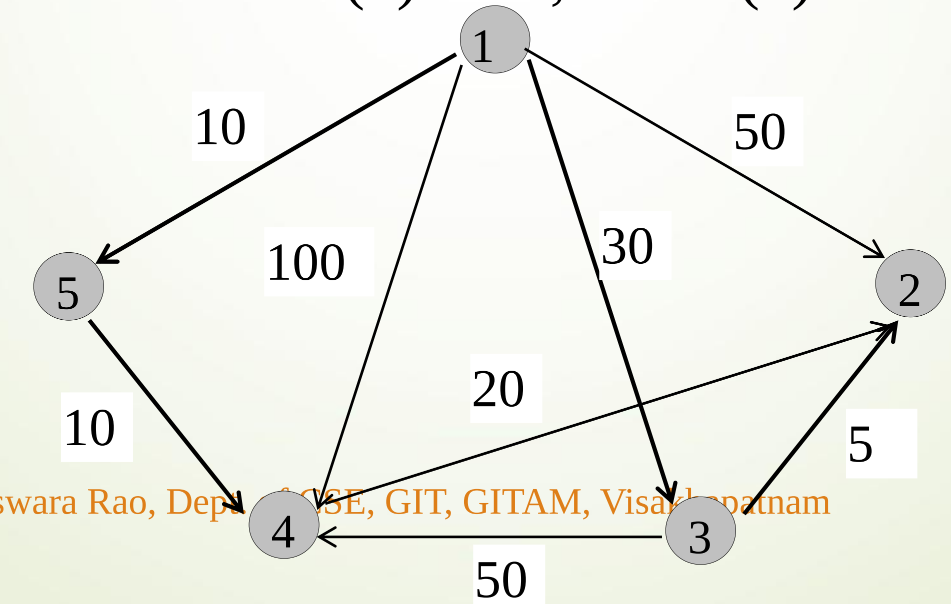
3. *Dist* (4) = 20, *From* (4) = 5



4. *Dist* (3) = 30, *From* (3) = 1



5. *Dist* (2) = 35, *From* (2) = 3



Shortest paths from source *1*

1 → 3 → 2 35

1 → 3 30

1 → 5 → 4 20

1 → 5 10

Dijkstra's Algorithm


```
shortest paths (int v, float cost[ ][size],float dist[ ],int n).
{
//dist[j],1≤j≤n is set to the length of the shortest path from v to j.
//dist[v] is set to zero.
//G is represented by its cost Adjacency matrix cost [1:n][1:n].
 int u; bool s[size];
        for(int i=1;i<=n;i++)
        {
        s[i]=false;
        dist[i]=cost[v][i];
        }
      s[v]=true;
      dist[v]=0.0;
      for(int num=2; num<n; num++)
       {
//choose u from among those vertexes not in such that dist[u] is minimum;
 s[u]=true;//put u in s;
 for(int w=1;w<=n;w++)
// for (each w adjacent to u with S[w]=false) do
       //update distances.
      If ((s[w]==false) && (dist[w]>dist[u]+cost[u][w]))
dist[w]=dist[u]+cost[u][w];
        }
}
```

3. Optimal Storage on Tapes

The problem:

Given n programs to be stored on tape, the lengths of these n programs are $l_1, l_2, \ldots, l_n$ respectively. Suppose the programs are stored in the order of $i_1, i_2, \ldots, i_n$

Let t_j be the time to retrieve program i_j.

Assume that the tape is initially positioned at the beginning.

t_j is proportional to the sum of all lengths of programs stored in front of the program i_j.

The goal is to minimize MRT (Mean Retrieval Time), $\frac{1}{n}\sum_{j=1}^{n} t_j$

i.e. want to minimize $\sum_{j=1}^{n}\sum_{k=1}^{j} l_{i_k}$

Ex: $n = 3, \quad (l_1, l_2, l_3) = (5,10,3)$

There are n! = 6 possible orderings for storing them.

	order	total retrieval time	MRT
1	1 2 3	5+(5+10)+(5+10+3)=38	38/3
2	1 3 2	5+(5+3)+(5+3+10)=31	31/3
3	2 1 3	10+(10+5)+(10+5+3)=43	43/3
4	2 3 1	10+(10+3)+(10+3+5)=41	41/3
5	3 1 2	3+(3+5)+(3+5+10)=29	29/3 ← Smallest
6	3 2 1	3+(3+10)+(3+10+5)=34	34/3

Note: The problem can be solved using greedy strategy, just always let the shortest program goes first.

(Can simply get the right order by using any sorting algorithm)

Analysis:

Try all combination: O(n!)

Shortest-length-First Greedy method: O ($n\log n$)

Shortest-length-First Greedy method:

Sort the programs s.t. $l_1 \le l_2 \le \ldots \le l_n$
and call this ordering *L*.

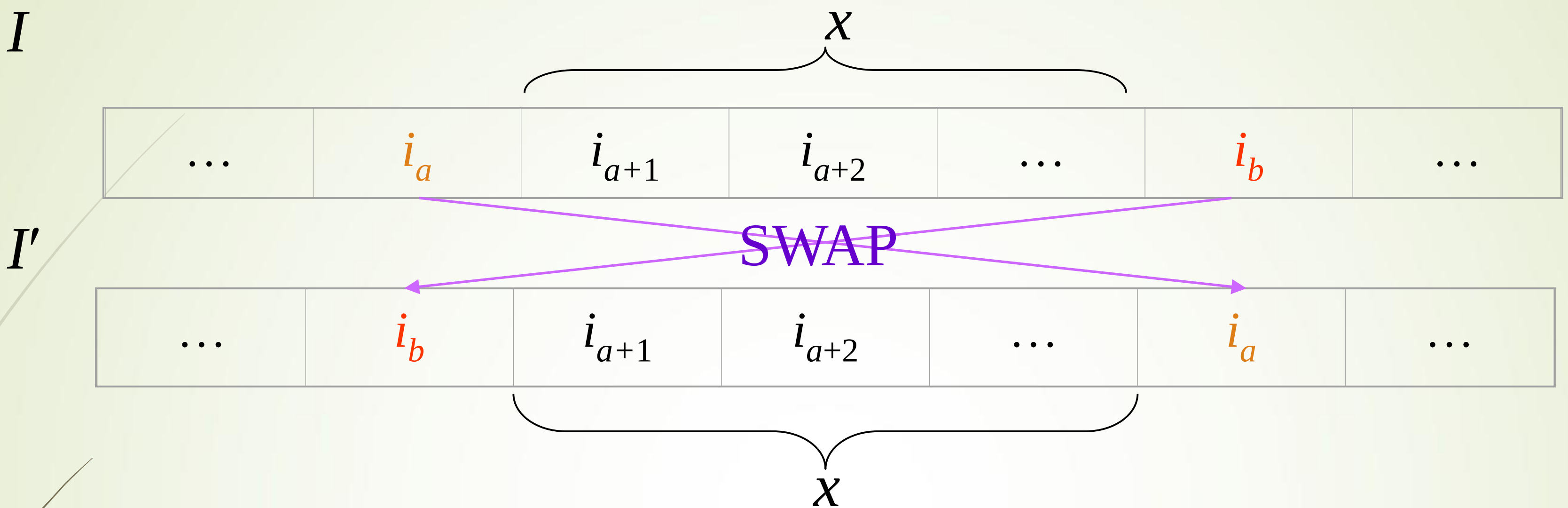
Next is to show that the ordering *L* is the best

Proof by contradiction:

Suppose Greedy ordering L is not optimal, then there exists some other permutation *I* that is optimal.

$I = (i_1, i_2, \ldots i_n)$ $\exists$ a < b, s.t. $l_{i_a} > l_{i_b}$ (otherwise $I = L$)

Interchange i_a and i_b in and call the new list I' :



In I', Program i_{a+1} will take less $(l_{i_a} - l_{i_b})$ time than in I to be retrieved

In fact, each program i_{a+1} , …, i_{b-1} will take less $(l_{i_a} - l_{i_b})$ time

For i_b, the retrieval time decreases $x + l_{i_a}$

For i_a, the retrieval time increases $x + l_{i_b}$

$$totalRT(I) - totalRT(I') = (b-a-1)(l_{i_a} - l_{i_b}) + (x + l_{i_a}) - (x + l_{i_b})$$

$$= (b-a)(l_{i_a} - l_{i_b}) > 0 \quad (\rightarrow\leftarrow)$$

Contradiction!!

Therefore, greedy ordering *L* is optimal