**Course Code**

CSC402

**Course Name**

Analysis of Algorithms

**Department of Computer Engineering**

AY 2021-2022

## Syllabus

**Module 3 Greedy Method Approach**

- General Method, Single source shortest path: Dijkstra Algorithm Fractional Knapsack problem, Job sequencing with deadlines, Minimum cost spanning trees: Kruskal and Prim’s algorithms

# Module 3 Greedy Method Approach

CE– SE–AOA

**Dr. Anil Kale**

Associate Professor

Dept. of Computer Engineering,

## General Method

### Overview

- 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.

## General Method

- 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.

## General Method

- 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 suboptimal 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 .

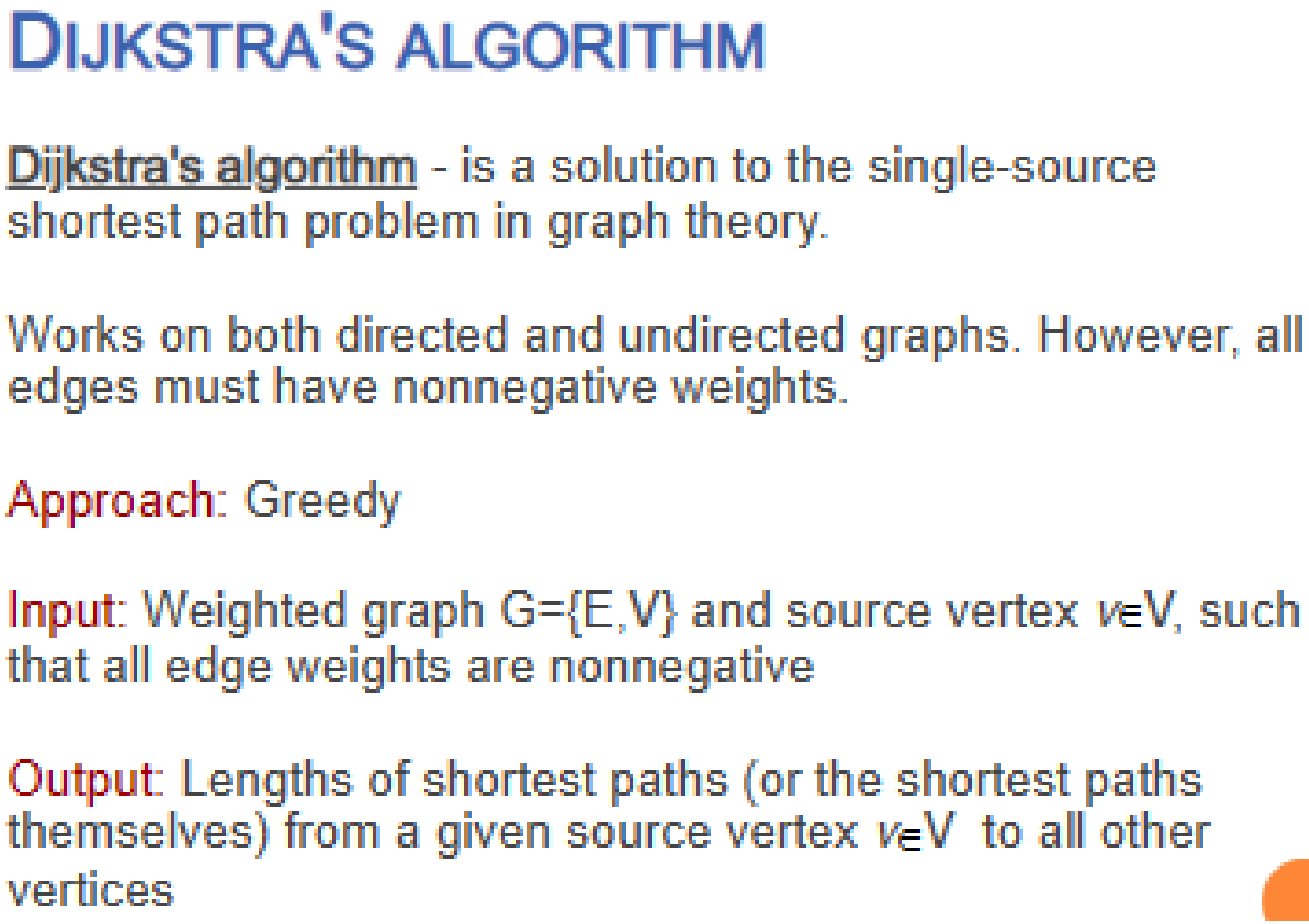
## Algorithm

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

## Single source shortest path: Dijkstra Algorithm

- 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 then (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.

## Single source shortest path: Dijkstra Algorithm



### DIJKSTRA'S ALGORITHM

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

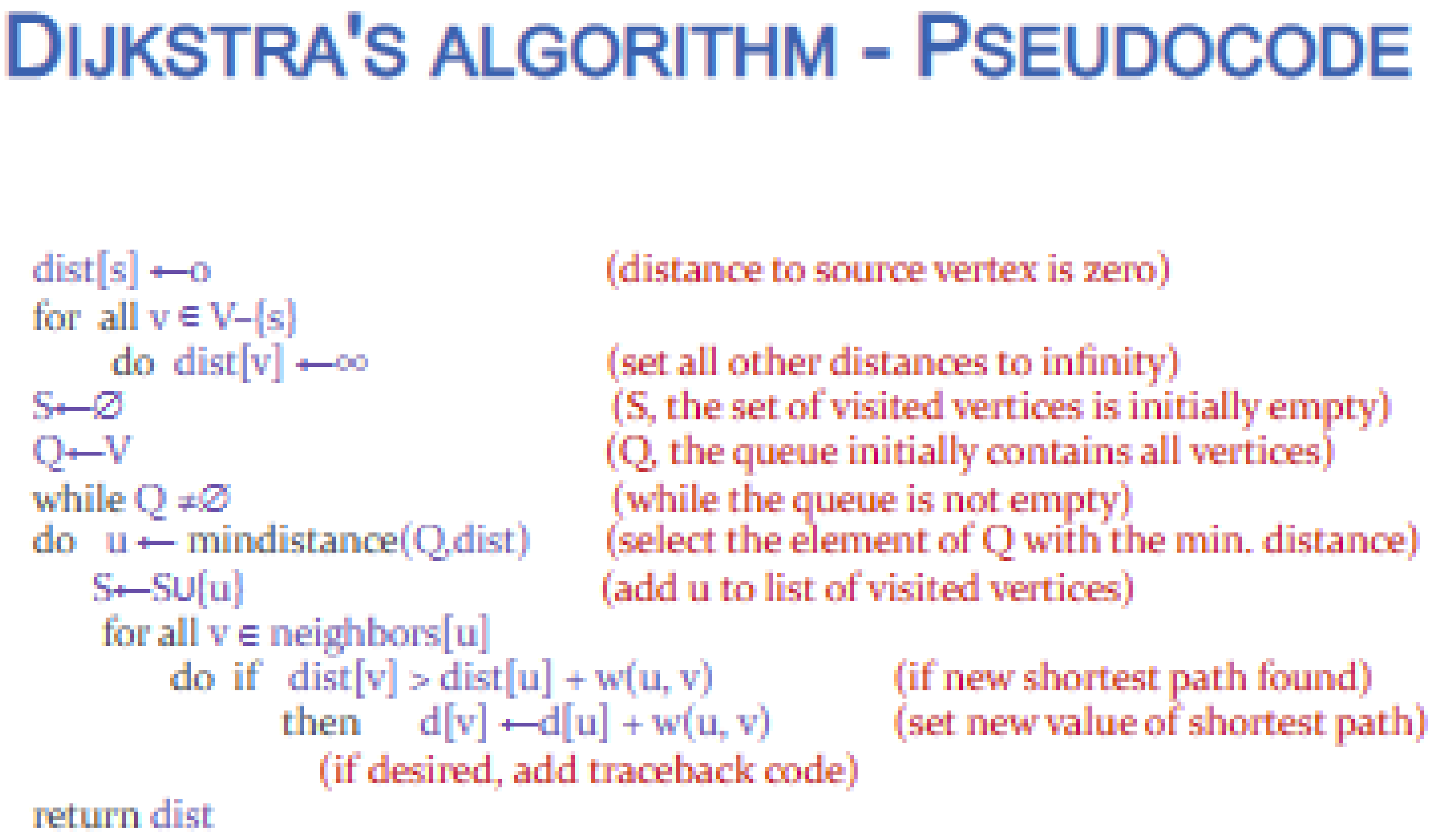
Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

# Single source shortest path: Dijkstra Algorithm



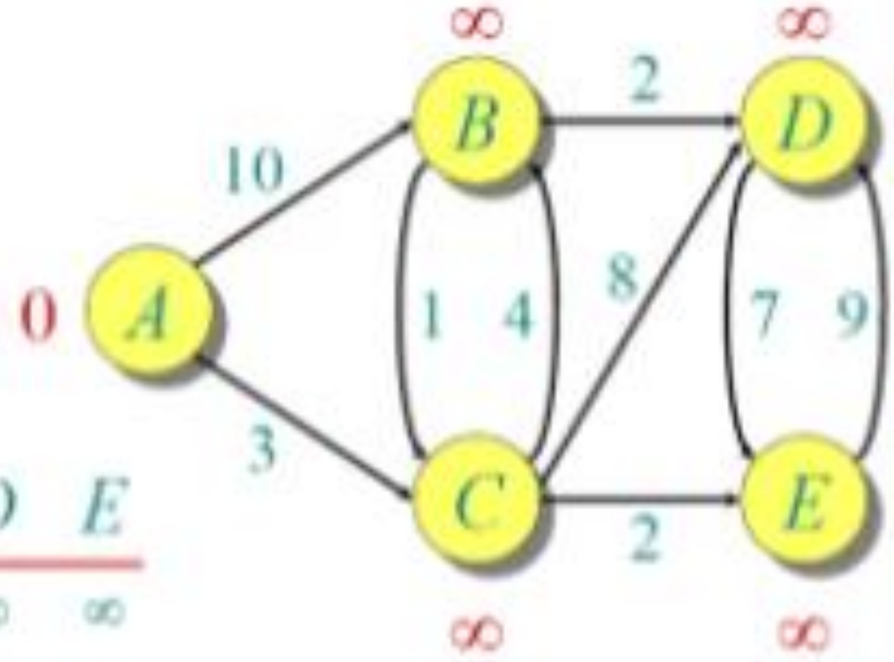
## DIJKSTRA'S ALGORITHM - PSEUDOCODE

```
dist[s] ←0                              (distance to source vertex is zero)
for  all v ∈ V–{s}
        do  dist[v] ←∞                  (set all other distances to infinity)
S←∅                                     (S, the set of visited vertices is initially empty)
Q←V                                     (Q, the queue initially contains all vertices)
while Q ≠∅                              (while the queue is not empty)
do   u ← mindistance(Q,dist)            (select the element of Q with the min. distance)
      S←S∪{u}                           (add u to list of visited vertices)
       for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                         then      d[v] ←d[u] + w(u, v)    (set new value of shortest path)
                          (if desired, add traceback code)
return dist
```

# Single source shortest path: Dijkstra Algorithm

## DIJKSTRA ANIMATED EXAMPLE

**Initialize:**

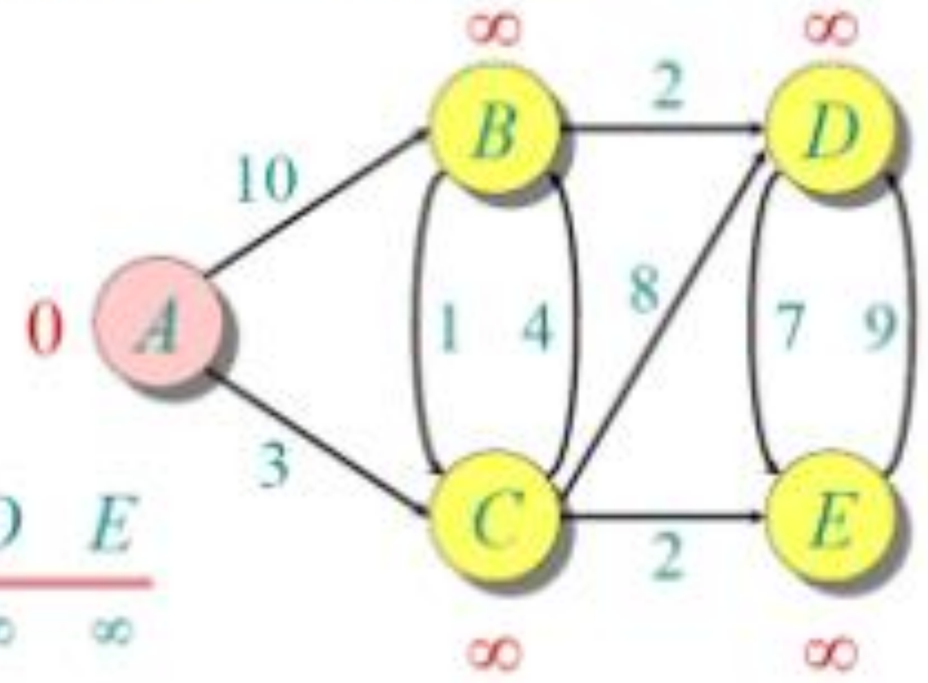


| $Q$: | $A$ | $B$ | $C$ | $D$ | $E$ |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |

$S$: {}

# Single source shortest path: Dijkstra Algorithm

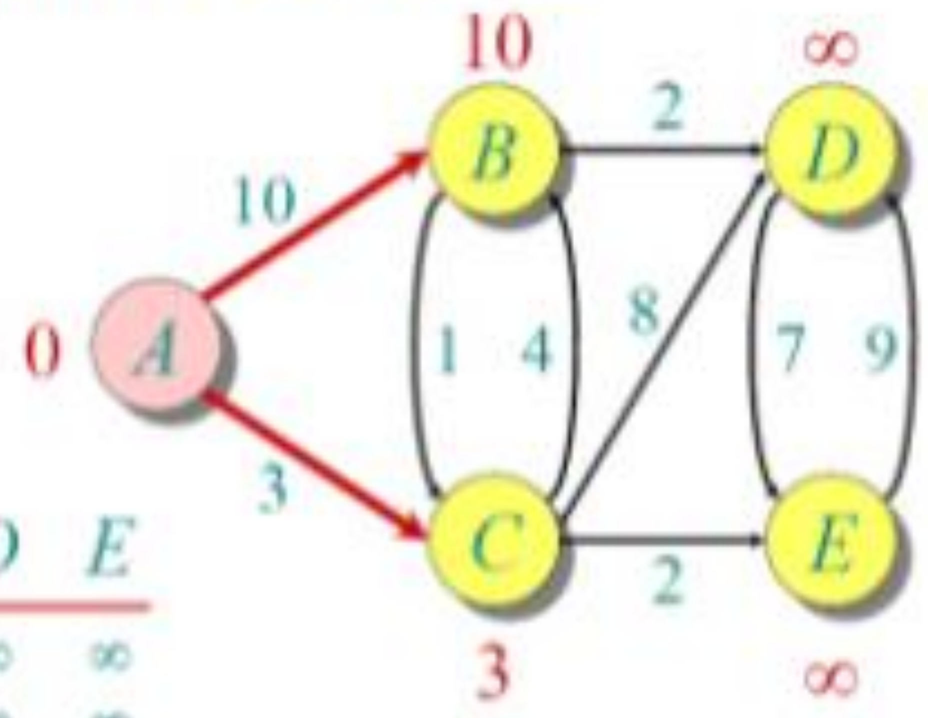
## DIJKSTRA ANIMATED EXAMPLE



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |

## Single source shortest path: Dijkstra Algorithm

### DIJKSTRA ANIMATED EXAMPLE

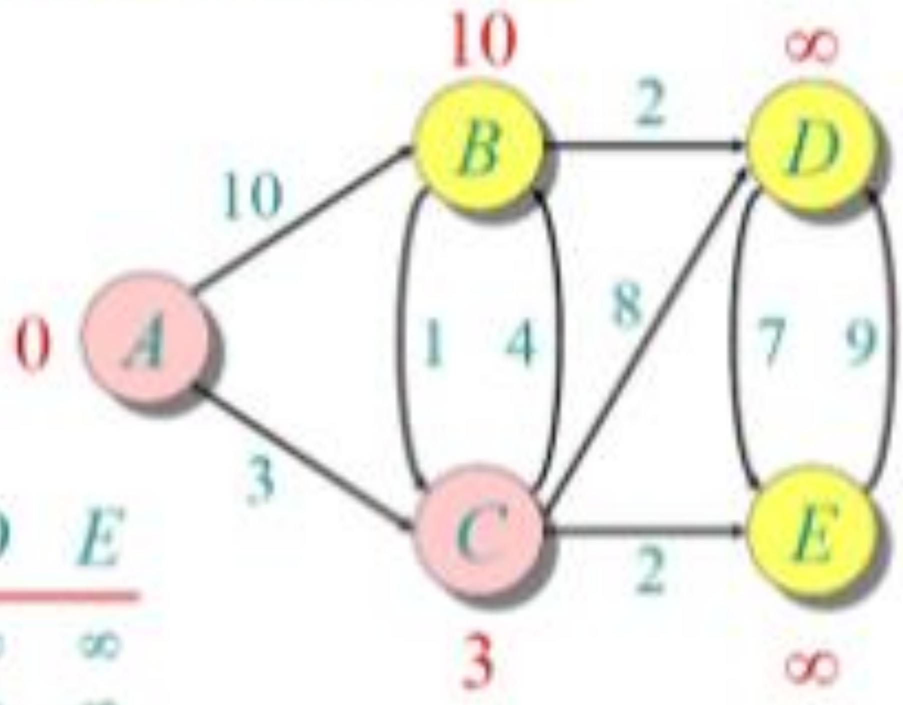


| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |

S: { A }

## Single source shortest path: Dijkstra Algorithm

### DIJKSTRA ANIMATED EXAMPLE

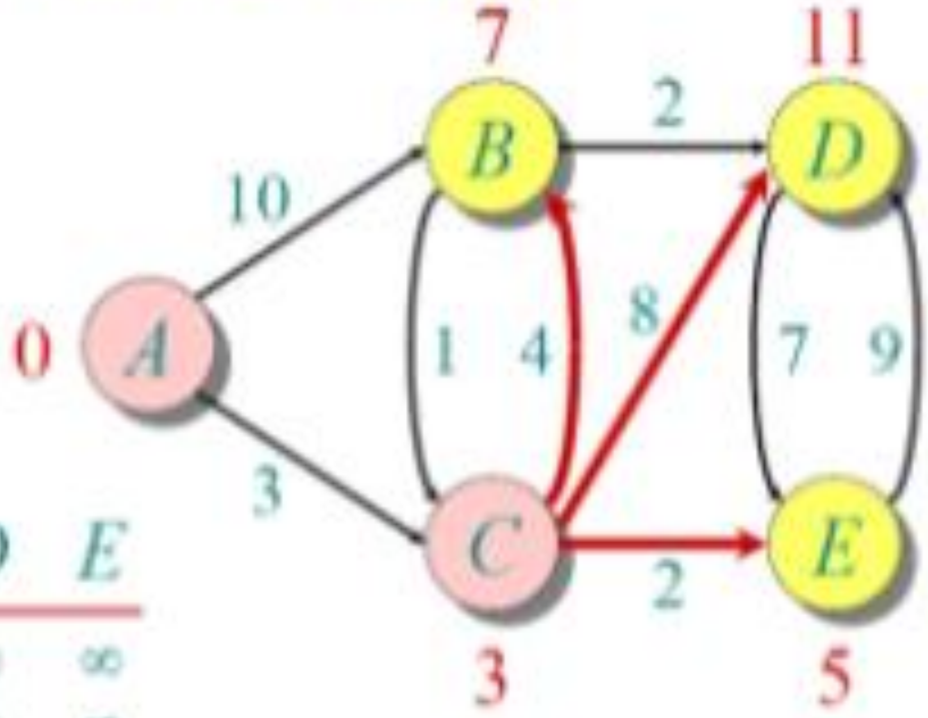


| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |

$S: \{ A, C \}$

## Single source shortest path: Dijkstra Algorithm

### DIJKSTRA ANIMATED EXAMPLE

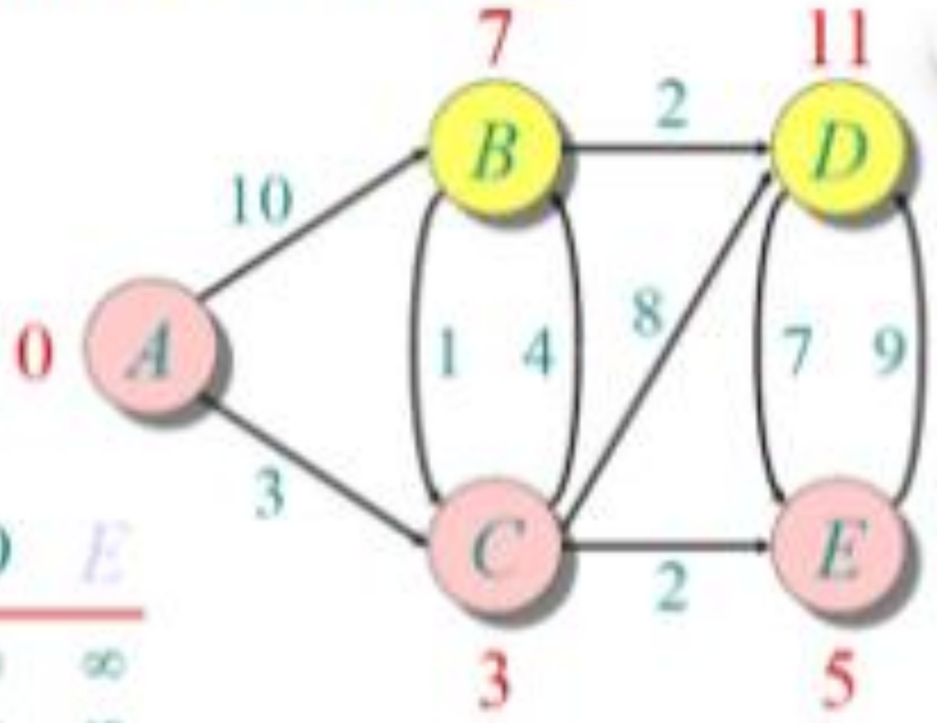


| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |

*S*: { *A*, *C* }

## Single source shortest path: Dijkstra Algorithm

### DIJKSTRA ANIMATED EXAMPLE

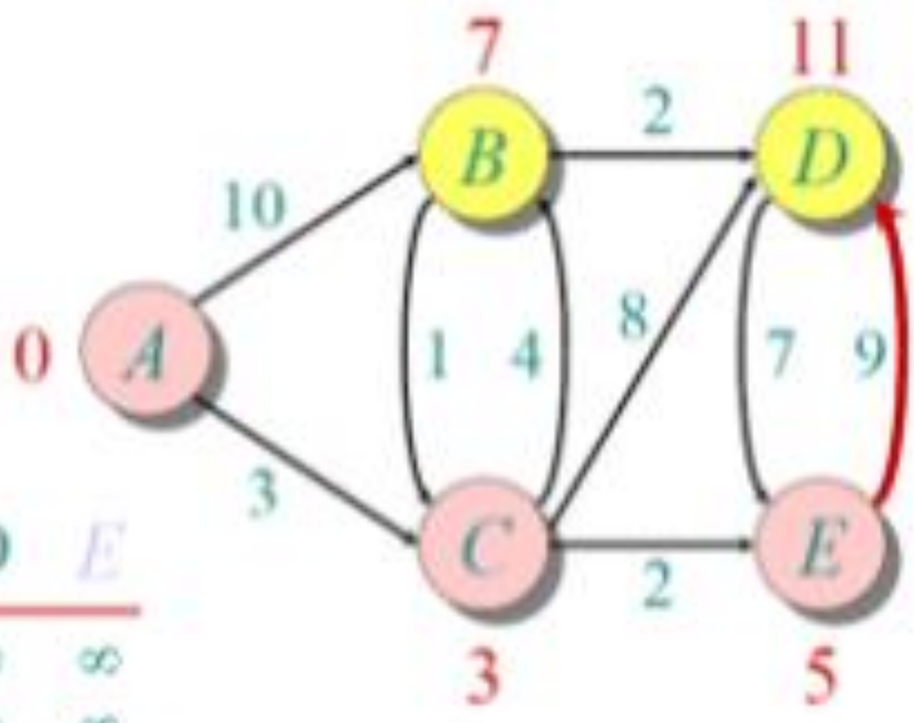


| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |

$S$: { $A$, $C$, $E$ }

# Single source shortest path: Dijkstra Algorithm

## DIJKSTRA ANIMATED EXAMPLE



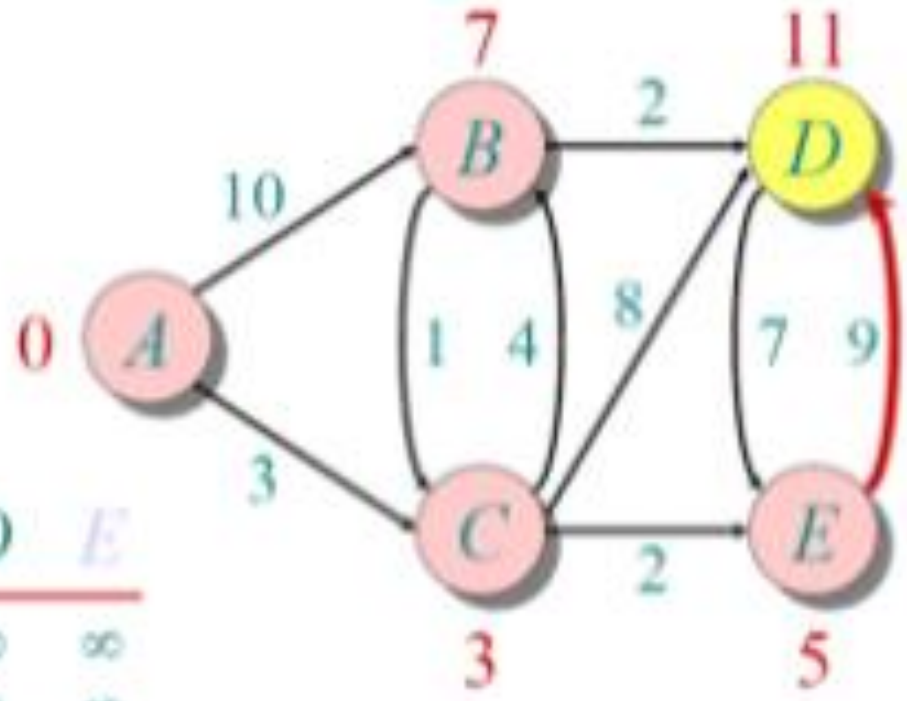
Q:

| A | B | C | D | E |
|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ |
|  | 10 | 3 | ∞ | ∞ |
|  | 7 |  | 11 | 5 |
|  | 7 |  | 11 |  |

S: { A, C, E }

# Single source shortest path: Dijkstra Algorithm

## DIJKSTRA ANIMATED EXAMPLE

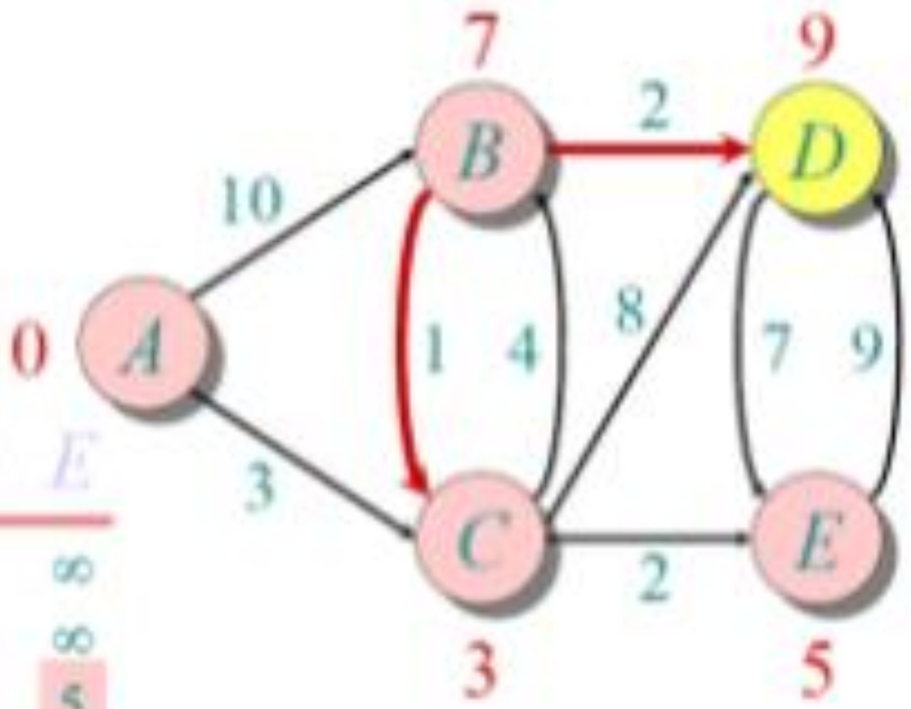


| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |
| | | 7 | | 11 | |

S: { A, C, E, B }

# Single source shortest path: Dijkstra Algorithm

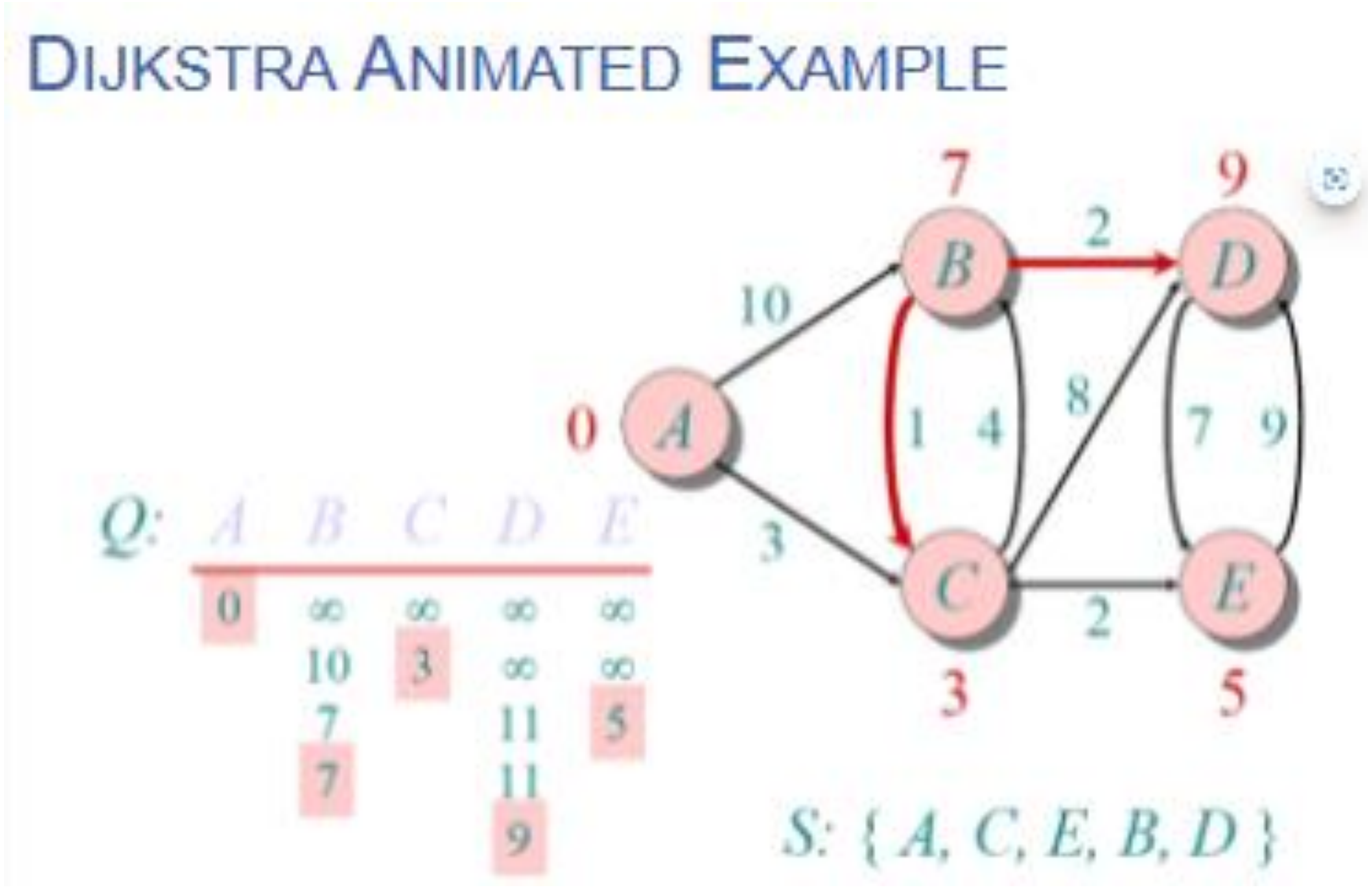
## DIJKSTRA ANIMATED EXAMPLE



| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |
| | | 7 | | 11 | |
| | | | | 9 | |

*S*: { *A*, *C*, *E*, *B* }

# Single source shortest path: Dijkstra Algorithm

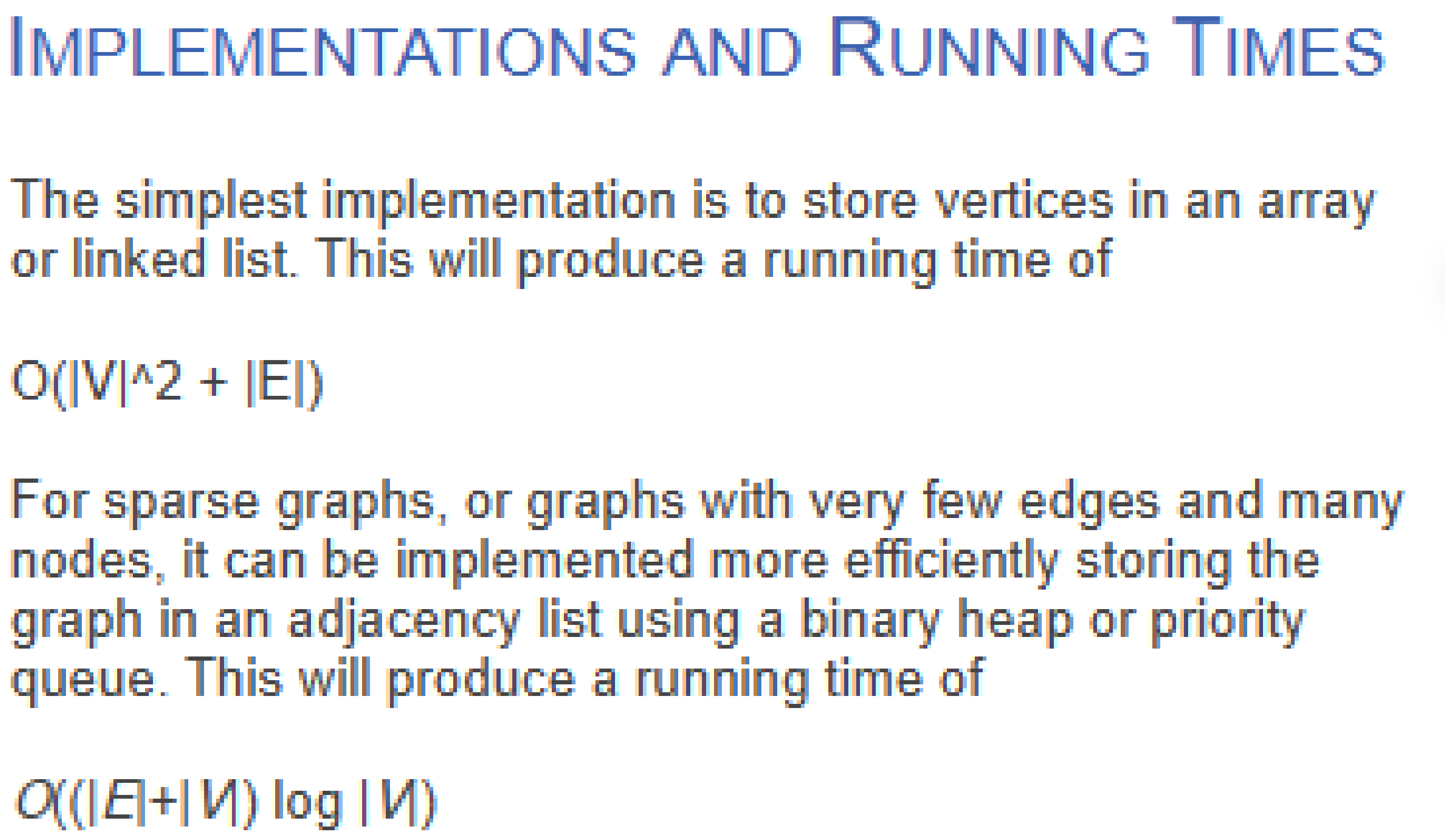


DIJKSTRA ANIMATED EXAMPLE

| Q: | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| | | 10 | 3 | ∞ | ∞ |
| | | 7 | | 11 | 5 |
| | | 7 | | 11 | |
| | | | | 9 | |

S: { A, C, E, B, D }

# Single source shortest path: Dijkstra Algorithm



## Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

O(|V|^2 + |E|)

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

## Single source shortest path: Dijkstra Algorithm

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

- Good for dense graphs (many edges)

- |V| vertices and |E| edges
- Initialization O(|V|)
- While loop O(|V|)
  - Find and remove min distance vertices O(|V|)
- Potentially |E| updates
  - Update costs O(1)

Total time $O(|V^2| + |E|) = O(|V^2|)$

## Single source shortest path: Dijkstra Algorithm

For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges) Dijkstra's implemented more efficiently by *priority queue*

- Initialization O(|V|) using O(|V|) buildHeap
- While loop O(|V|)
  - Find and remove min distance vertices O(log |V|) using O(log |V|) deleteMin

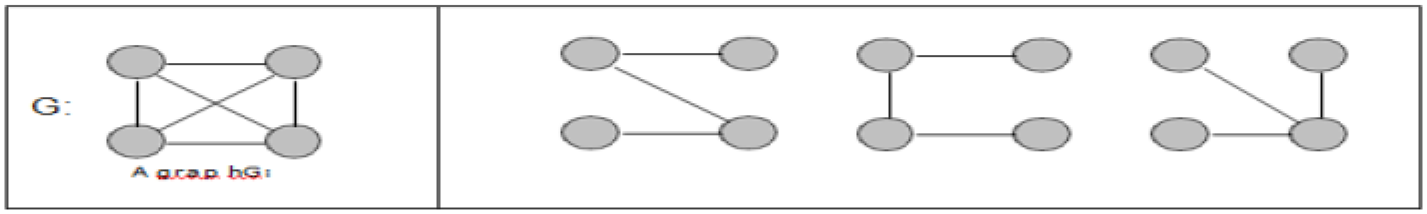
- Potentially |E| updates
  - Update costs O(log |V|) using decreaseKey

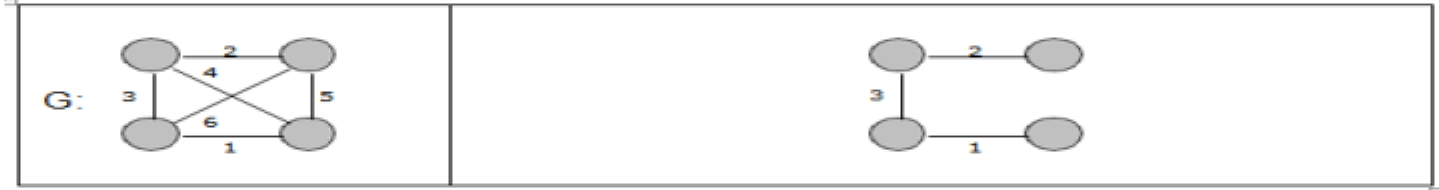
Total time O(|V|log|V| + |E|log|V|) = O(|E|log|V|)

- |V| = O(|E|) assuming a connected graph

## Minimum cost spanning trees: Kruskal and Prim’s algorithms

- 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 graphG: Theminimalspanningtreefromweightedgraph G:

## Minimum cost spanning trees: Kruskal and Prim’s algorithms

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

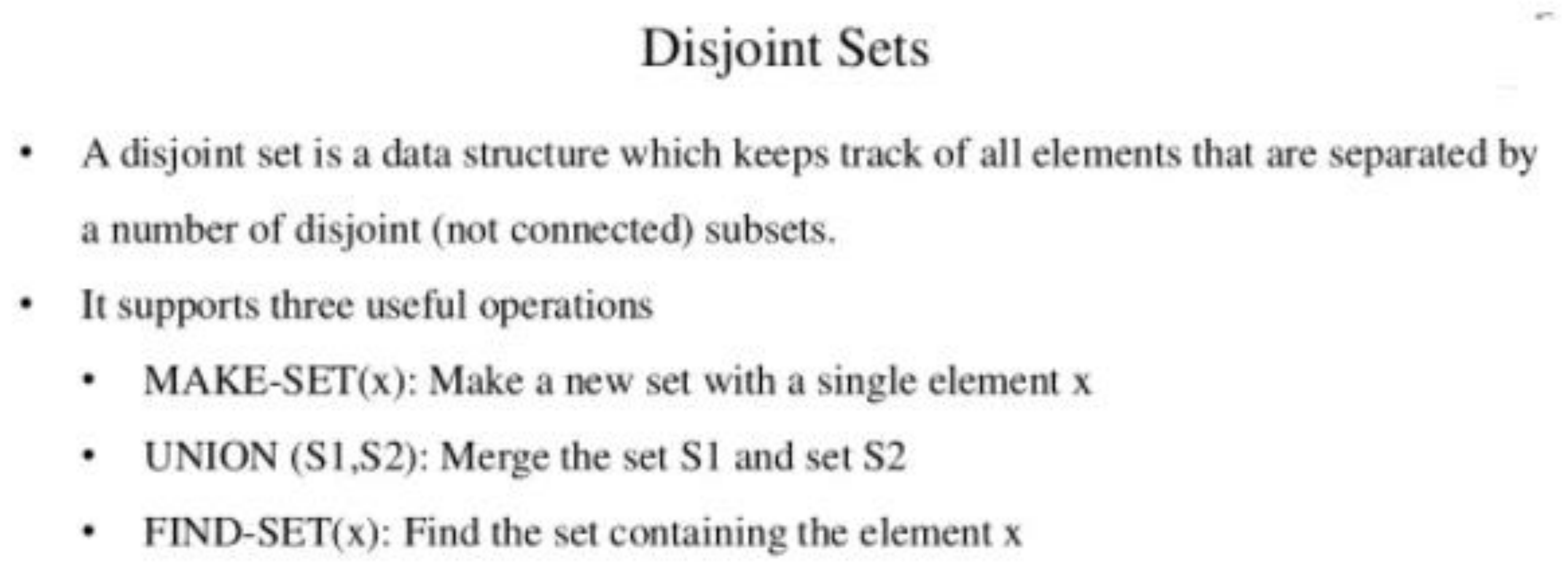
1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used 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.

## Minimum cost spanning trees: Kruskal and Prim’s algorithms

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

## MST : Kruskal algorithms



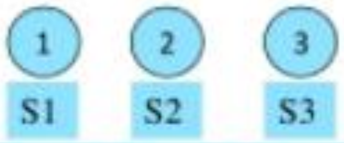
### Disjoint Sets

- A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets.
- It supports three useful operations
  - MAKE-SET(x): Make a new set with a single element x
  - UNION (S1,S2): Merge the set S1 and set S2
  - FIND-SET(x): Find the set containing the element x

## MST : Kruskal algorithms

### Disjoint Sets

- A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets.
- It supports three useful operations
  - MAKE-SET(x): Make a new set with a single element x
  - UNION (S1,S2): Merge the set S1 and set S2
  - FIND-SET(x): Find the set containing the element x

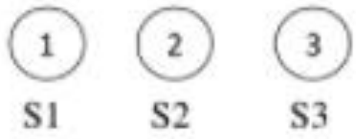


MAKE-SET(1), MAKE-SET(2), MAKE-SET(3) creates new set S1,S2,S3

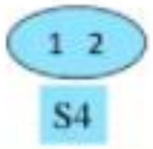
## MST : Kruskal algorithms

### Disjoint Sets

- A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets.
- It supports three useful operations
  - MAKE-SET(x): Make a new set with a single element x
  - UNION (S1,S2): Merge the set S1 and set S2
  - FIND-SET(x): Find the set containing the element x



MAKE-SET(1), MAKE-SET(2), MAKE-SET(3) creates new set S1,S2,S3

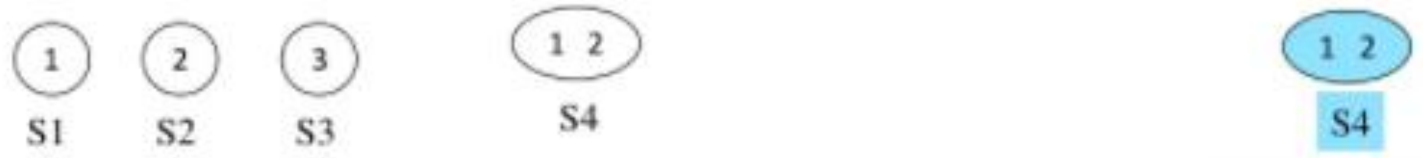


UNION (1,2) merge set S1 and set S2 to create set S4

## MST : Kruskal algorithms

### Disjoint Sets

- A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets
- It supports three useful operations
  - MAKE-SET(x): Make a new set with a single element x
  - UNION (S1,S2): Merge the set S1 and set S2
  - FIND-SET(x): Find the set containing the element x



MAKE-SET(1), MAKE-SET(2), MAKE-SET(3) creates new set S1,S2,S3

UNION (1,2) merge set S1 and set S2 to create set S4

FIND-SET(2) returns set S4

## MST : Kruskal algorithms

### Kruskal's Algorithm

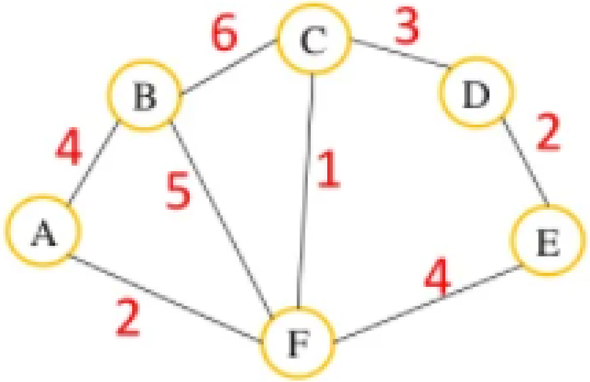
```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
            A = A $\cup$ $\{(v_1,v_2)\}$
            UNION($v_1,v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ $\{(v_1,v_2)\}$
            UNION($v_1,v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```

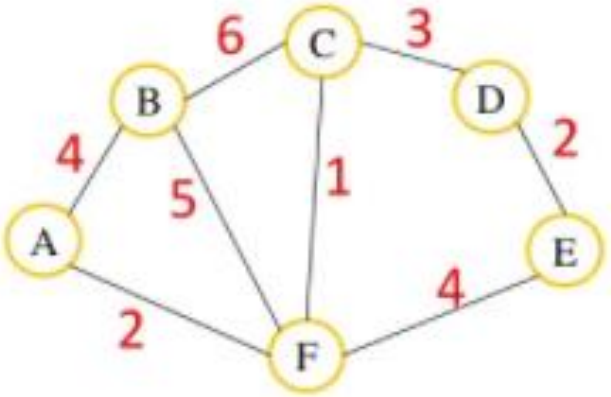


| Edges | Weight |
|---|---|
| AB | 4 |
| BC | 6 |
| CD | 3 |
| DE | 2 |
| EF | 4 |
| AF | 2 |
| BF | 5 |
| CF | 1 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
                A = A ∪ {($v_1$,$v_2$)}
                UNION($v_1$,$v_2$)
        else
                Remove edge ($v_1$,$v_2$)
    return A
```



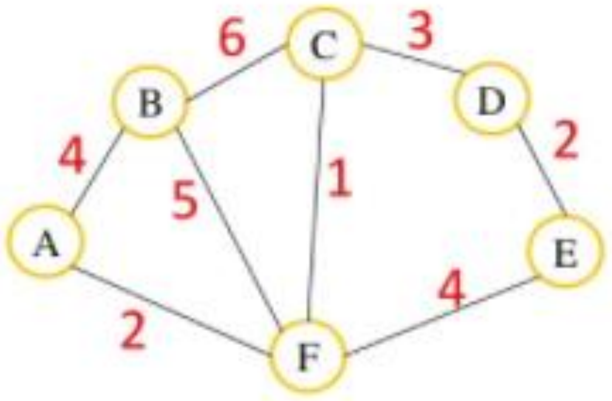
A ={ }

| Edges | Weight |
|---|---|
| AB | 4 |
| BC | 6 |
| CD | 3 |
| DE | 2 |
| EF | 4 |
| AF | 2 |
| BF | 5 |
| CF | 1 |

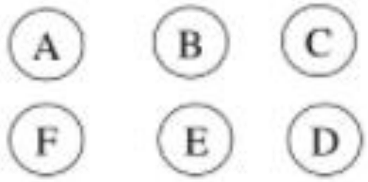
# MST : Kruskal algorithms

KRUSKAL(V,E):

```
A = ∅
foreach $v$ ∈ V:
    MAKE-SET($v$)
Sort E by weight increasingly
foreach ($v_1$,$v_2$) ∈ E:
    if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {($v_1$,$v_2$)}
            UNION($v_1$,$v_2$)
    else
            Remove edge ($v_1$,$v_2$)
return A
```



A ={ }

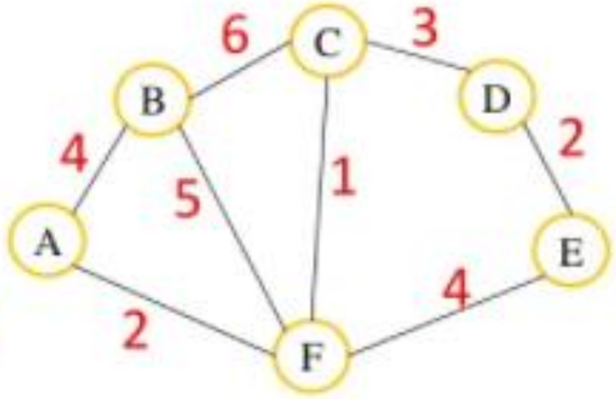


| Edges | Weight |
|---|---|
| AB | 4 |
| BC | 6 |
| CD | 3 |
| DE | 2 |
| EF | 4 |
| AF | 2 |
| BF | 5 |
| CF | 1 |

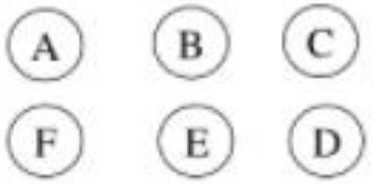
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1$,$v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



A ={ }

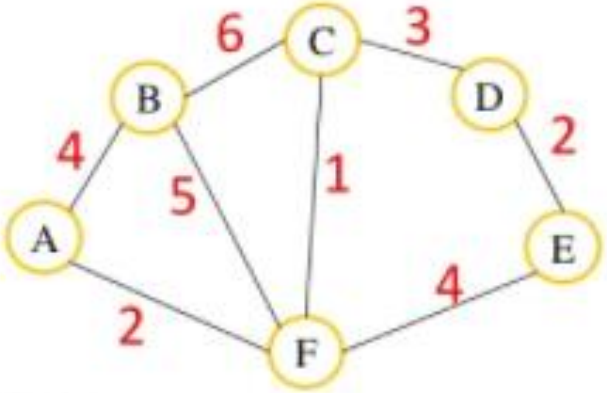


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

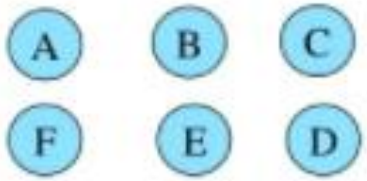
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach ($v_1$,$v_2$) ∈ E:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {($v_1$,$v_2$)}
                UNION($v_1$,$v_2$)
        else
                Remove edge ($v_1$,$v_2$)
    return A
```



A ={ }

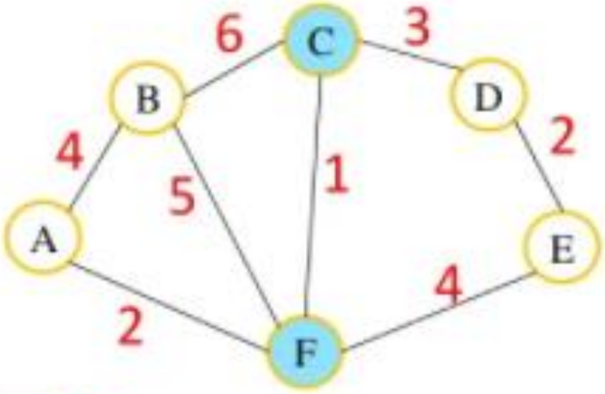


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

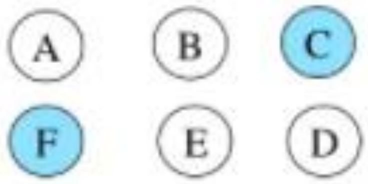
## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {$(v_1,v_2)$}
            UNION($v_1,v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```



A ={ }

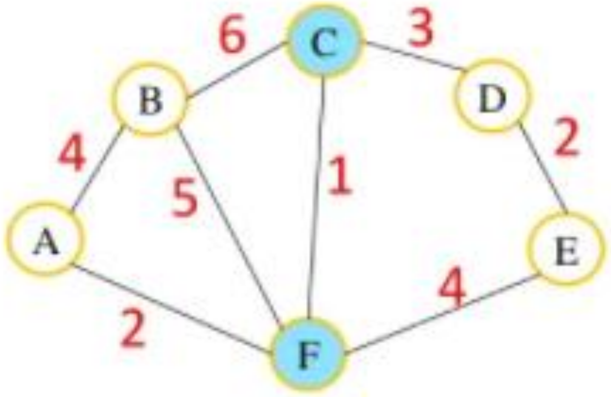


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

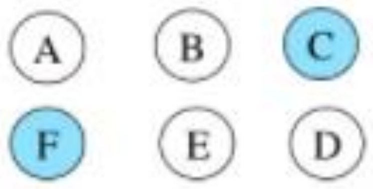
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            $A = A \cup \{(v_1,v_2)\}$
            UNION($v_1,v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F)}

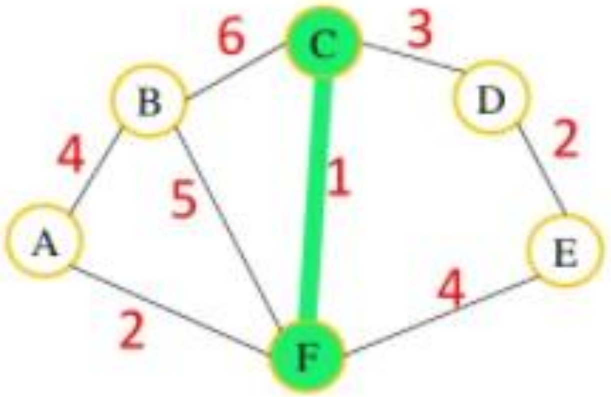


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

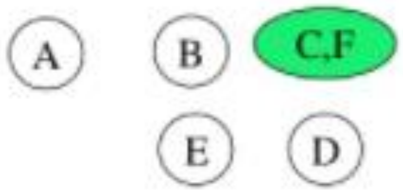
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1$,$v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F)}

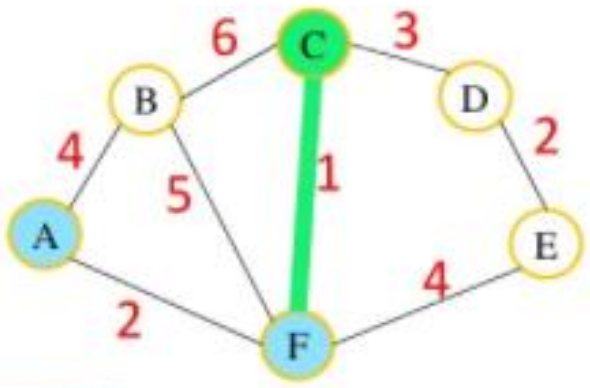


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

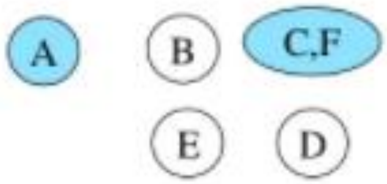
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
            A = A ∪ {$(v_1,v_2)$}
            UNION($v_1$,$v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F)}

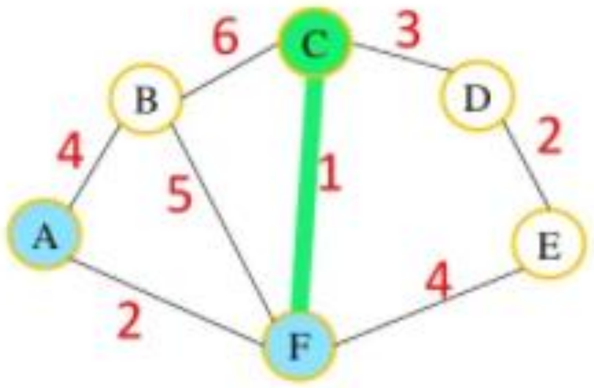


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

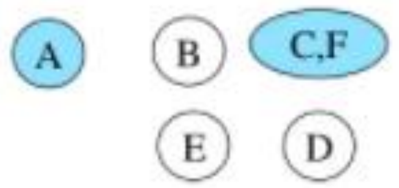
## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {($v_1$,$v_2$)}
            UNION($v_1$,$v_2$)
        else
            Remove edge ($v_1$,$v_2$)
    return A
```



A ={(C,F),(A,F)}

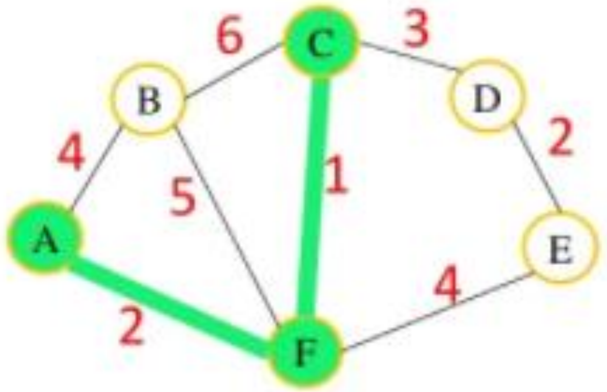


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

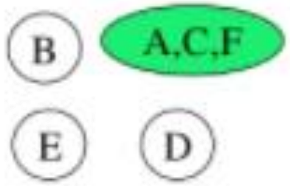
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {$(v_1,v_2)$}
            UNION($v_1$,$v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F),(A,F)}

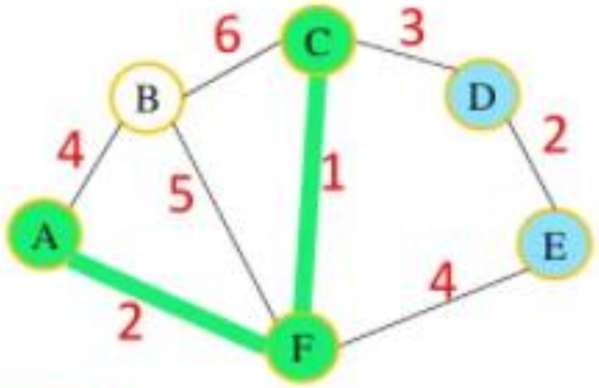


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

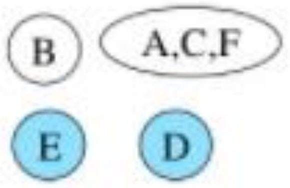
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {($v_1$,$v_2$)}
            UNION($v_1$,$v_2$)
        else
            Remove edge ($v_1$,$v_2$)
    return A
```



A ={(C,F),(A,F)}

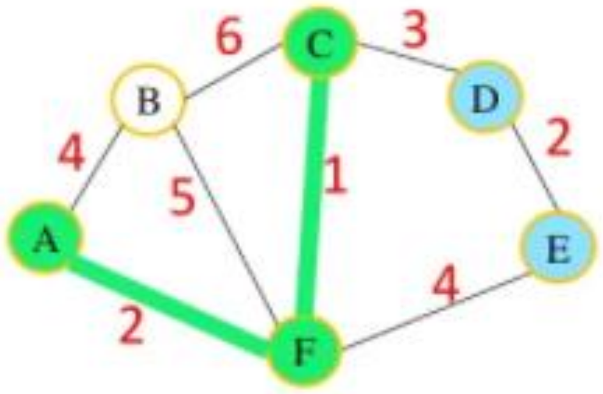


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

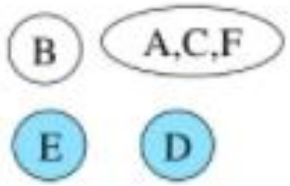
```
KRUSKAL(V,E):

    A = ∅
    foreach $v$ ∈ V:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach ($v_1$,$v_2$) ∈ E:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {($v_1$,$v_2$)}
            UNION($v_1$,$v_2$)
        else
            Remove edge ($v_1$,$v_2$)
    return A
```



A ={(C,F),(A,F),(D,E)}

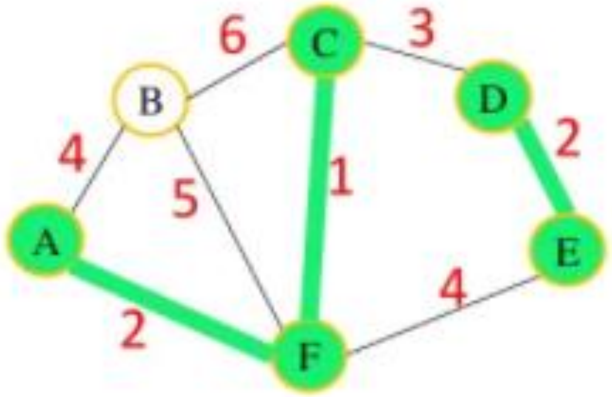
| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |



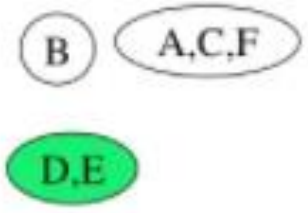
## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach v ∈ V:
        MAKE-SET(v)
    Sort E by weight increasingly
    foreach (v_1,v_2) ∈ E:
        if FIND-SET(v_1) ≠ FIND-SET(v_2):
                A = A ∪ {(v_1,v_2)}
                UNION(v_1,v_2)
        else
                Remove edge (v_1,v_2)
    return A
```



A ={(C,F),(A,F),(D,E)}

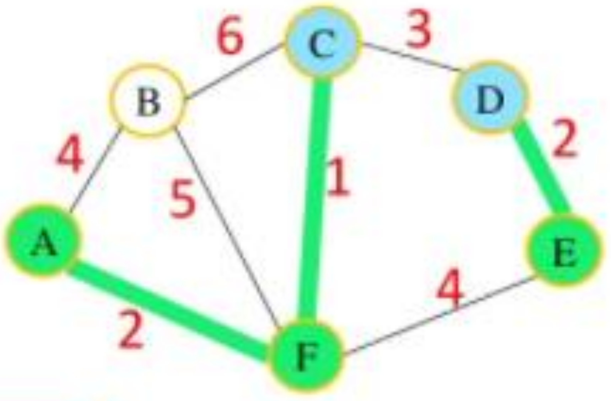


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

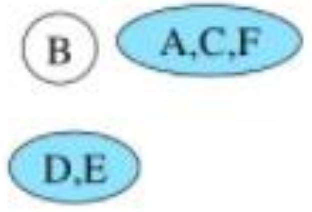
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ $\{(v_1,v_2)\}$
                UNION($v_1$,$v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F),(A,F),(D,E)}

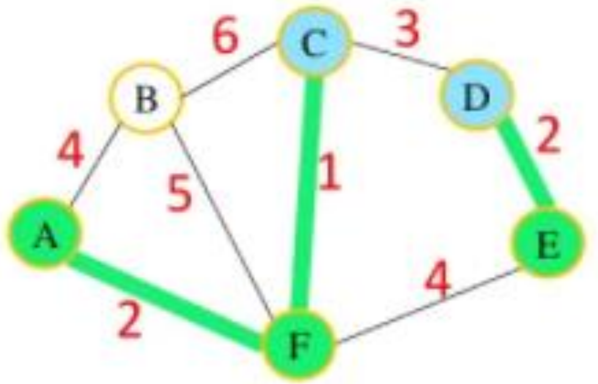


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

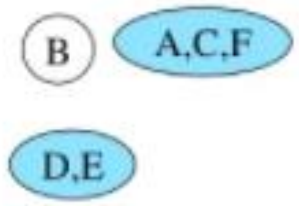
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {($v_1$,$v_2$)}
            UNION($v_1$,$v_2$)
        else
            Remove edge ($v_1$,$v_2$)
    return A
```



A ={(C,F),(A,F),(D,E), (C,D)}

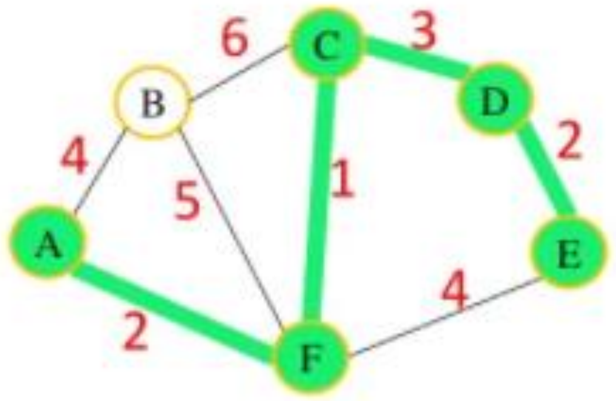


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

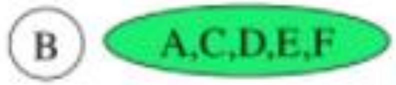
## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1$,$v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F),(A,F),(D,E), (C,D)}

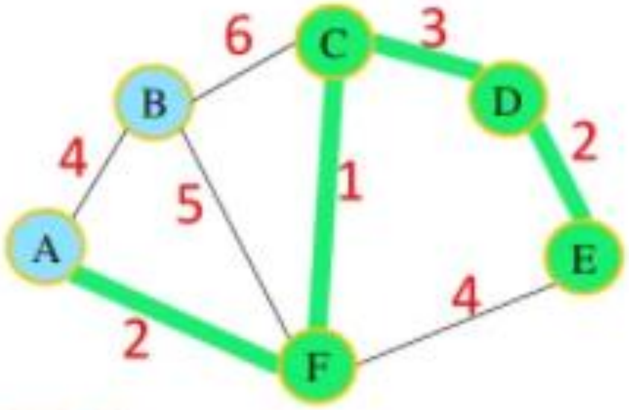


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

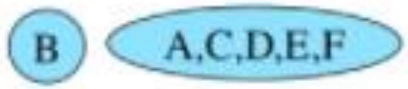
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {$(v_1,v_2)$}
            UNION($v_1$,$v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F),(A,F),(D,E),
(C,D)}

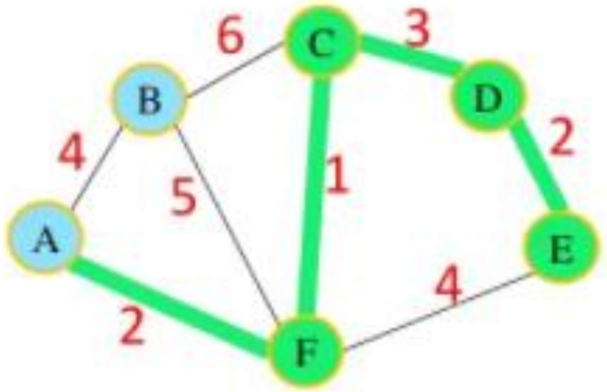


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

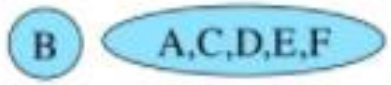
# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {($v_1$,$v_2$)}
            UNION($v_1$,$v_2$)
        else
            Remove edge ($v_1$,$v_2$)
    return A
```



A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

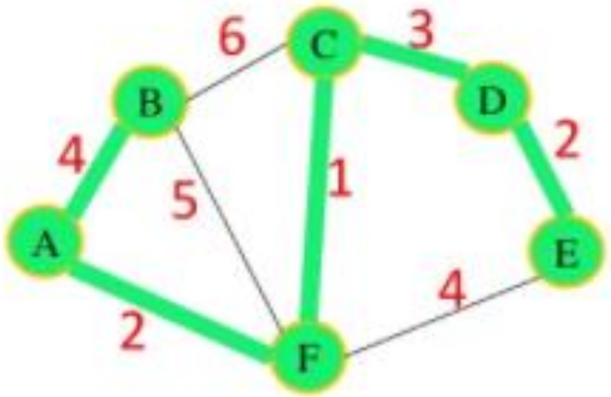


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
            A = A ∪ {$(v_1,v_2)$}
            UNION($v_1$,$v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```



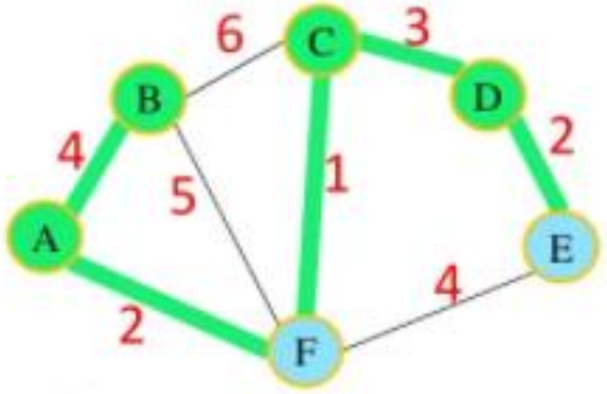
A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach ($v_1$,$v_2$) ∈ E:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {($v_1$,$v_2$)}
                UNION($v_1$,$v_2$)
        else
                Remove edge ($v_1$,$v_2$)
    return A
```



A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

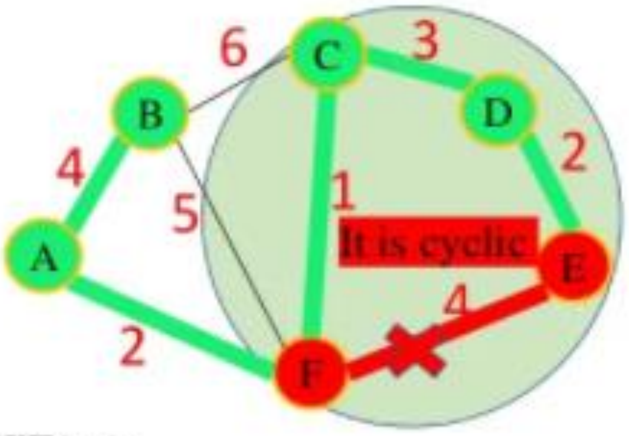
A,B,C,D,E,F

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1,v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

A,B,C,D,E,F

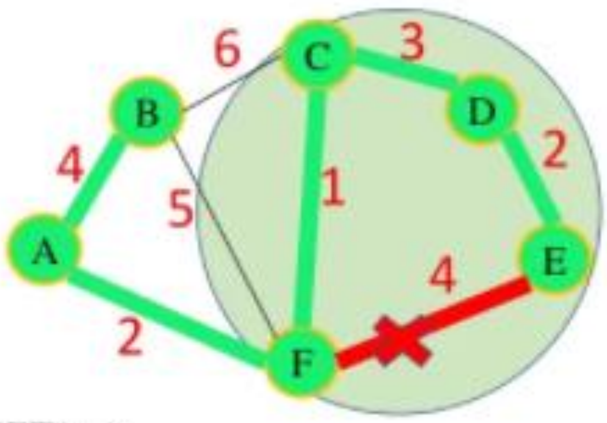
Is in a same set.

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {$(v_1,v_2)$}
            UNION($v_1,v_2$)
        else
            Remove edge ($v_1,v_2$)
    return A
```



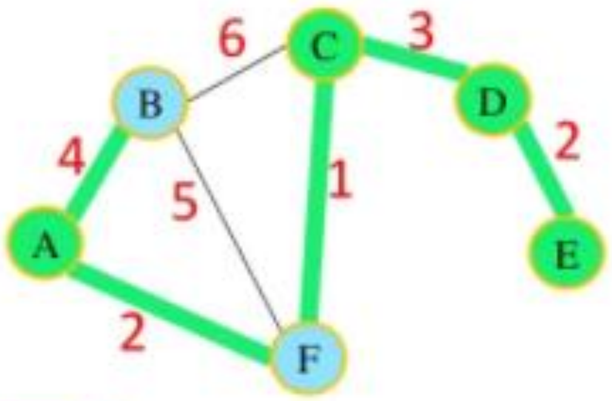
A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
            A = A ∪ {$(v_1,v_2)$}
            UNION($v_1,v_2$)
        else
            Remove edge $(v_1,v_2)$
    return A
```



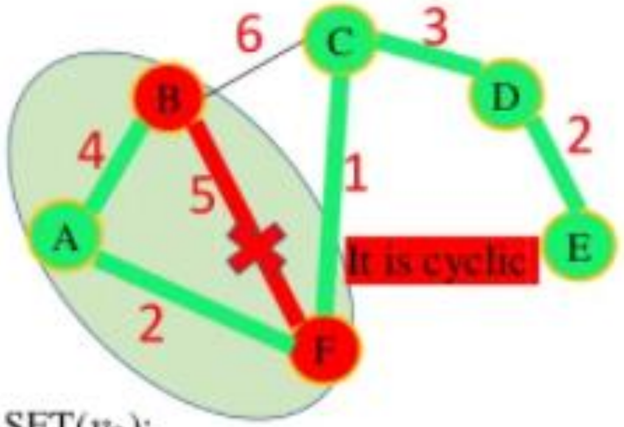
A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1,v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



$$A = \{(C,F),(A,F),(D,E),(C,D),(A,B)\}$$

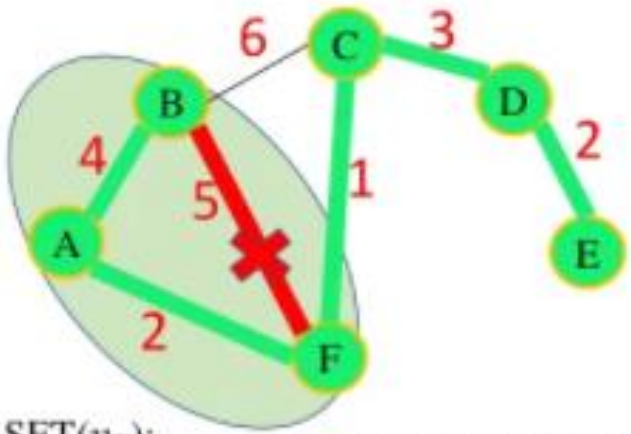
A,B,C,D,E,F
Is in a same set.

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1,v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



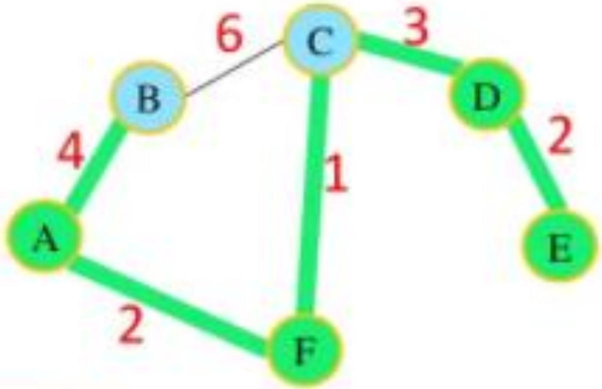
A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1,v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



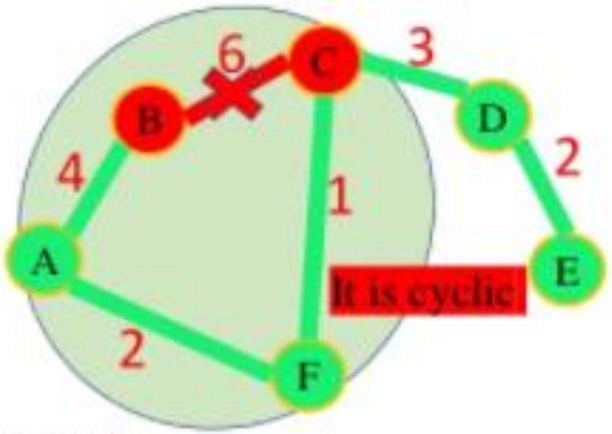
A ={(C,F),(A,F),(D,E),(C,D),(A,B)}

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v$ ∈ V:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach ($v_1$,$v_2$) ∈ E:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {($v_1$,$v_2$)}
                UNION($v_1$,$v_2$)
        else
                Remove edge ($v_1$,$v_2$)
    return A
```



A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

A,B,C,D,E,F

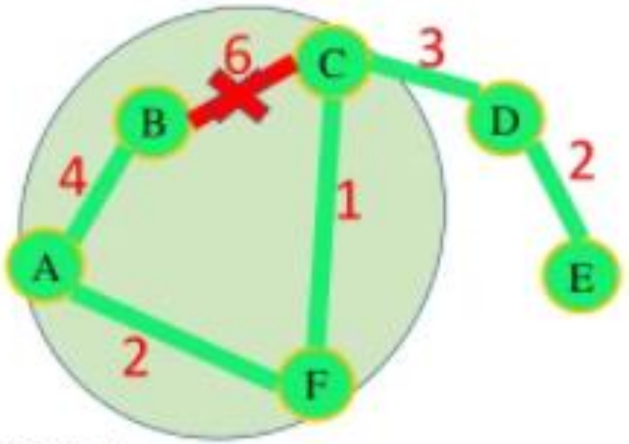
Is in a same set.

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

## MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) ≠ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1,v_2$)
        else
                Remove edge ($v_1,v_2$)
    return A
```



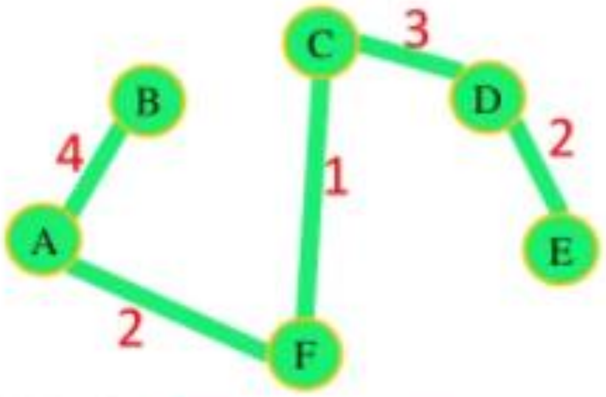
A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
                A = A ∪ $\{(v_1,v_2)\}$
                UNION($v_1,v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```



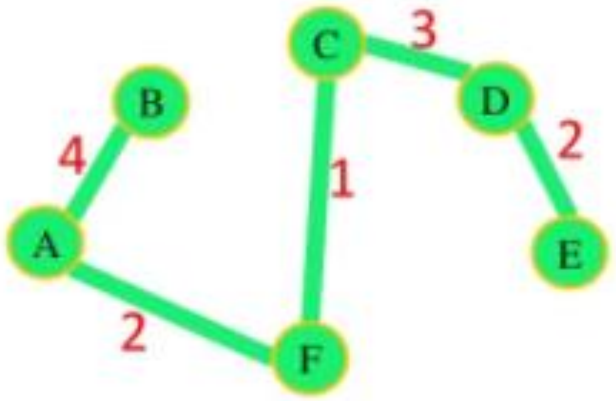
A ={(C,F),(A,F),(D,E),
(C,D),(A,B)}

| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

# MST : Kruskal algorithms

```
KRUSKAL(V,E):

    A = ∅
    foreach $v \in V$:
        MAKE-SET($v$)
    Sort E by weight increasingly
    foreach $(v_1,v_2) \in E$:
        if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
                A = A ∪ {$(v_1,v_2)$}
                UNION($v_1,v_2$)
        else
                Remove edge $(v_1,v_2)$
    return A
```

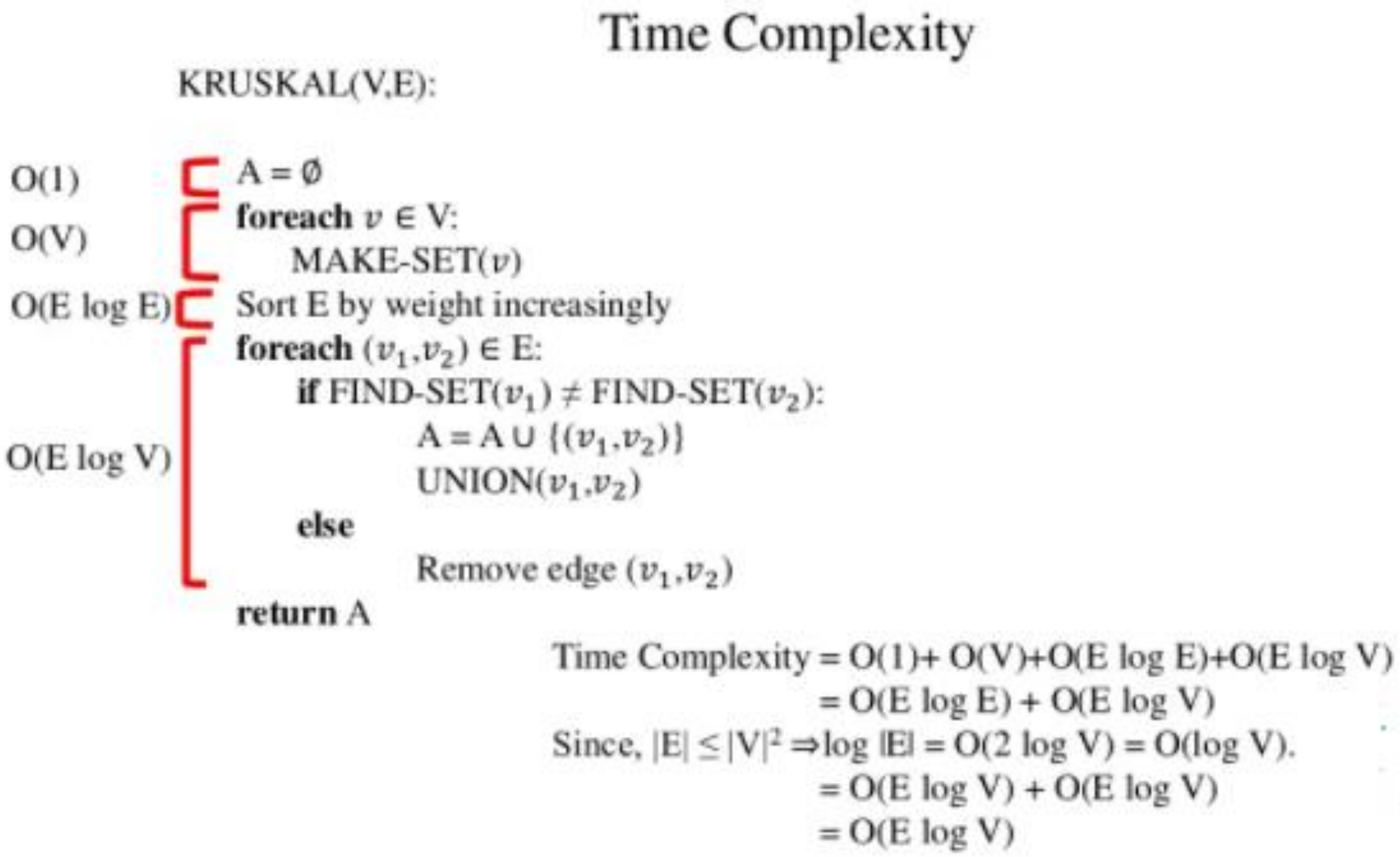


| Edges | Weight |
|---|---|
| CF | 1 |
| AF | 2 |
| DE | 2 |
| CD | 3 |
| AB | 4 |
| FE | 4 |
| BF | 5 |
| BC | 6 |

$$A = \{ (C,F),(A,F),(D,E),(C,D),(A,B) \}$$

Total Weight :- 1 + 2 + 2 + 3 + 4
= 12

## MST : Kruskal algorithms



### Time Complexity

```
              KRUSKAL(V,E):

O(1)            $A = \emptyset$
O(V)            foreach $v \in V$:
                    MAKE-SET($v$)
O(E log E)      Sort E by weight increasingly
O(E log V)      foreach $(v_1,v_2) \in E$:
                    if FIND-SET($v_1$) $\neq$ FIND-SET($v_2$):
                        $A = A \cup \{(v_1,v_2)\}$
                        UNION($v_1,v_2$)
                    else
                        Remove edge $(v_1,v_2)$
                return A
```

Time Complexity $= O(1) + O(V) + O(E \log E) + O(E \log V)$

$= O(E \log E) + O(E \log V)$

Since, $|E| \leq |V|^2 \Rightarrow \log |E| = O(2 \log V) = O(\log V)$.

$= O(E \log V) + O(E \log V)$

$= O(E \log V)$

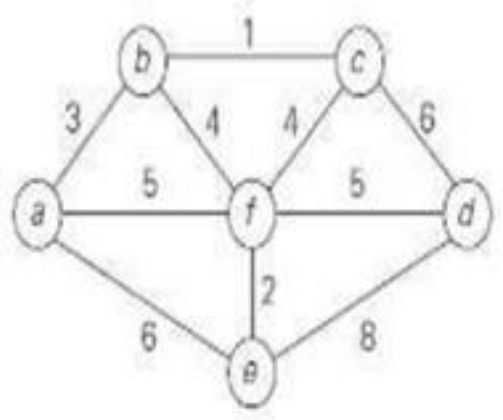
# MST : Kruskal algorithms

```
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 reheapify using Adjust;
        $j :=$ Find($u$); $k :=$ Find($v$);
        if ($j \neq k$) then
        {
            $i := i + 1$;
            $t[i, 1] := u$; $t[i, 2] := v$;
            $mincost := mincost + cost[u, v]$;
            Union($j, k$);
        }
    }
    if ($i \neq n - 1$) then write ("No spanning tree");
    else return $mincost$;
}
```

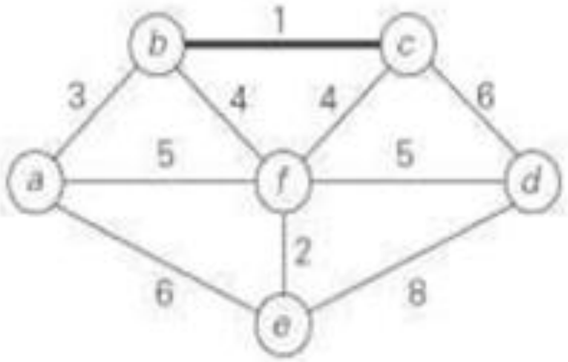
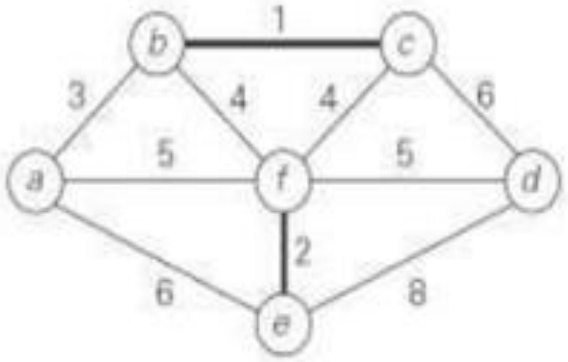
## MST : Kruskal algorithms

**Illustration**

An example of Kruskal's algorithm is shown below. The selected edges are shown in bold.

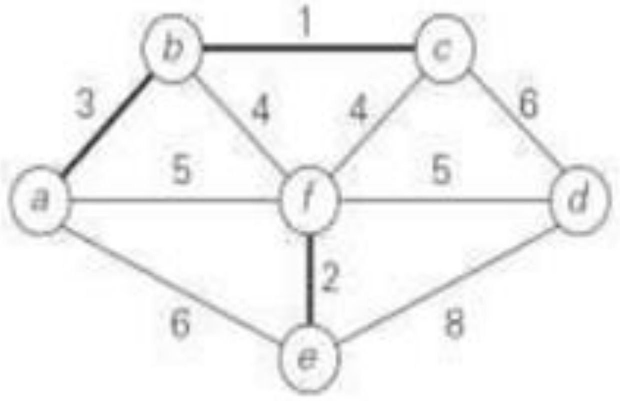


## MST : Kruskal algorithms

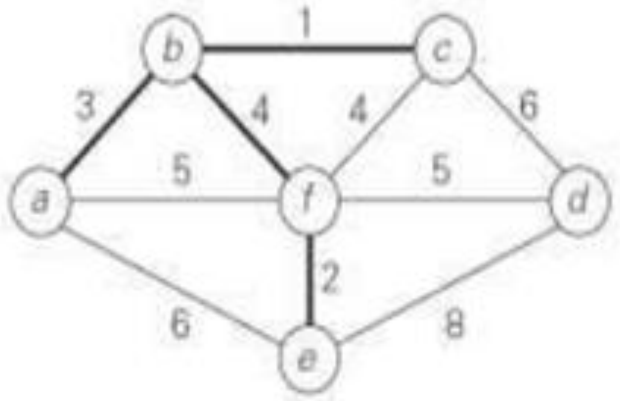
| Tree edges | Sorted list of edges | Illustration |
|---|---|---|
| | **bc** ef ab bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |
| bc<br>1 | bc **ef** ab bf cf af df ae cd de<br>1 2 3 4 4 5 5 6 6 8 |  |

## MST : Kruskal algorithms

| ef | bc | ef | **ab** | bf | cf | af | df | ae | cd | de |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 8 |



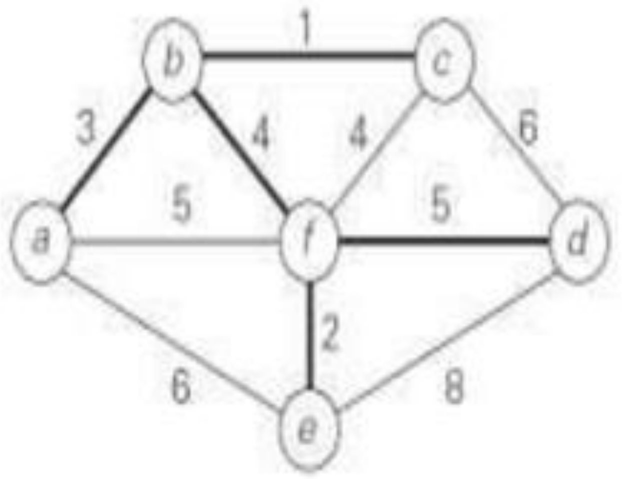
| ab | bc | ef | ab | **bf** | cf | af | df | ae | cd | de |
|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 8 |



# MST : Kruskal algorithms

| bf |
|---|
| 4 |

| bc | ef | ab | bf | cf | af | **df** | ae | cd | de |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 6 | 8 |



| df |
|---|
| 5 |

# MST : Prim’s algorithms

Algorithm Prim (E, cost, n,t)

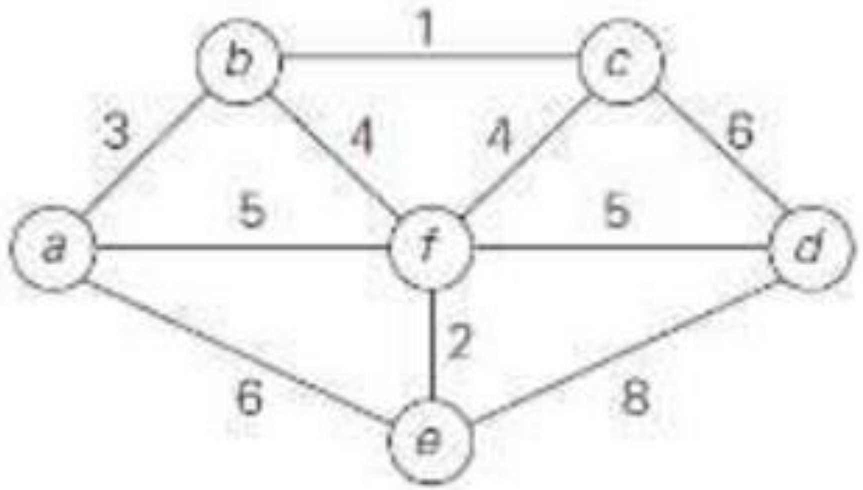
- // E is the set of edges in G. cost [1:n, 1:n] is thecost
- // adjacency matrix of an n vertex graph such that cost [i, j]is
- // either a positive real number or µif no edge (i, j)exists.
- // A minimum spanning tree is computed and stored as a setof
- // edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edgein
- // the minimum-cost spanning tree. The final cost isreturned.
- {
- Let (k, l) be an edge of minimum cost inE; mincost := cost [k,l];
- t [1, 1] := k; t [1, 2] :=l;

## MST : Prim’s algorithms

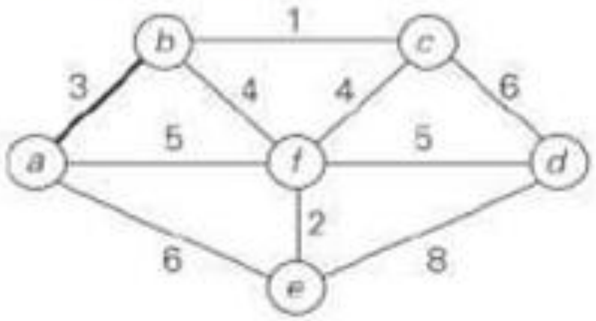
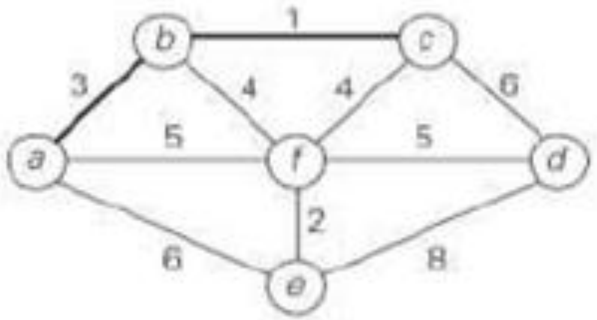
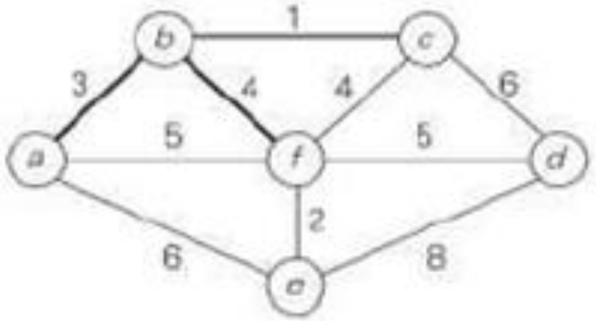
- for   i :=1 tondo              //Initializenearif  (cost [i, l] < cost [i, k]) then near [i] :=l;
- else near [i] := k; near [k] :=near [l] :=0;
- for  i:=2 to n - 1do          // Find n - 2 additional edges fort.
- {
- Let j be an index such that near [j] $^1$0**and**
- cost [j, near [j]] isminimum;
- t [i, 1] := j; t [i, 2] := near [j]; mincost := mincost + cost [j, near [j]]; near [j] :=0
- for   k:= 1 tondo              // Updatenear[].
- if ((near [k] $^1$0) **and** (cost [k, near [k]] > cost [k, j])) then near [k] :=j;
- }
- returnmincost;
- }

## MST : Prim's algorithms

**Example:** An example of prim's algorithm is shown below. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.

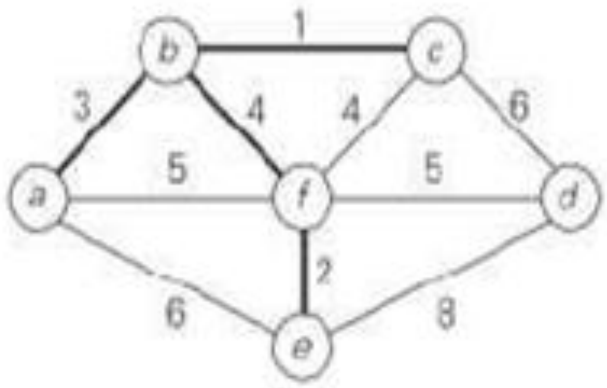


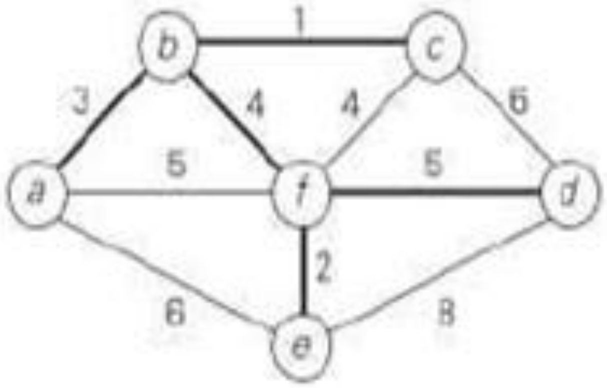
# MST : Prim's algorithms

| *Tree vertices* | *Remaining vertices* | *Illustration* |
|---|---|---|
| a(−, −) | **b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5) |  |
| b(a, 3) | **c(b, 1)** d(−, ∞) e(a, 6) f(b, 4) |  |
| c(b, 1) | d(c, 6) e(a, 6) **f(b, 4)** |  |

# MST : Prim's algorithms

| | |
|---|---|
| f(b, 4) | d(f, 5) **e(f, 2)** |
| e(f, 2) | **d(f, 5)** |
| d(f, 5) | |





## MST : Prim’s algorithms

**Analysis of Efficiency**

The efficiency of Prim’s algorithm depends on the data structures chosen for the **graph** itself and for the **priority queue** of the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

1. If a graph is represented by its **weight matrix** and the priority queue is implemented as an **unordered array**, the algorithm’s running time will be in **$\Theta(|V|^2)$**. Indeed, on each of the $|V| - 1$ iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

## MST : Prim’s algorithms

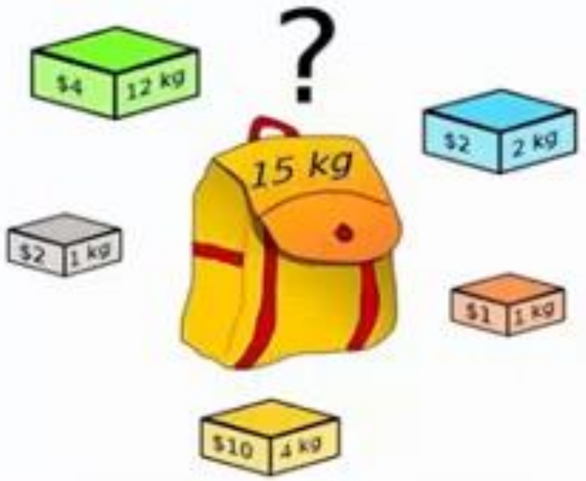
We can implement the priority queue as a min-heap. (A min-heap is a complete binary tree in which every element is less than or equal to its children.) Deletion of the smallest element from and insertion of a new element into a min-heap of size n are O(log n) operations.

2. If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in O(|E| log |V |).

This is because the algorithm performs |V| − 1 deletions of the smallest element and makes |E| verifications and, possibly, changes of an element’s priority in a min-heap of size not exceeding |V|. Each of these operations, as noted earlier, is a O(log |V|) operation. Hence, the running time of this implementation of Prim’s algorithm is in (|V| − 1+ |E|) O (log |V |) = O(|E| log |V |) because, in a connected graph, |V| − 1≤ |E|.

## Fractional Knapsack problem

# Knapsack problem



- **We are given n objects and a knapsack.**

  For i = 1, 2, .. ., n, object i has a ***positive weight*** ***wi*** and a ***positive value*** ***vi***.

- The knapsack can carry a weight **not exceeding** **W**. ***Our aim is to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint.***

24

## Fractional Knapsack problem

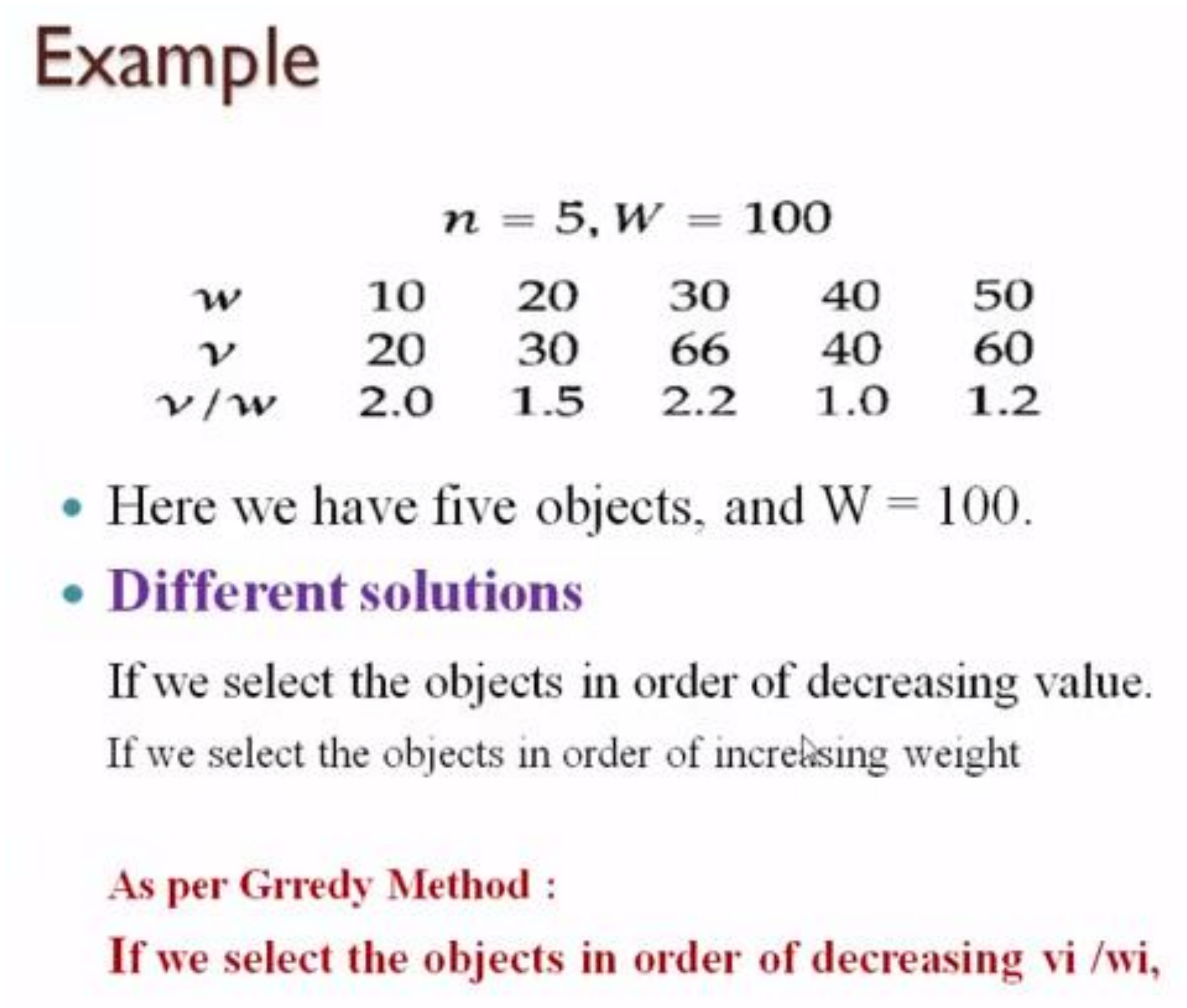
- In this first version of the problem we assume that the objects can be broken into smaller pieces, so we may decide to carry only a fraction xi of object i, where 0 < **xi** < **1.**

- In this case, object i contributes xiwi to the total weight in the knapsack, and xivi, to the value of the load. In symbols, the problem can be stated as follows:

$$\text{maximize} \sum_{i=1}^{n} x_i v_i \quad \text{subject to} \sum_{i=1}^{n} x_i w_i \leq W$$

- *where vi > 0, wi > 0 and 0 <= xi <= 1 for 1 <= i <= n.*

## Fractional Knapsack problem



Example

$$n = 5, W = 100$$

| | | | | | |
|---|---|---|---|---|---|
| $w$ | 10 | 20 | 30 | 40 | 50 |
| $v$ | 20 | 30 | 66 | 40 | 60 |
| $v/w$ | 2.0 | 1.5 | 2.2 | 1.0 | 1.2 |

- Here we have five objects, and W = 100.
- **Different solutions**

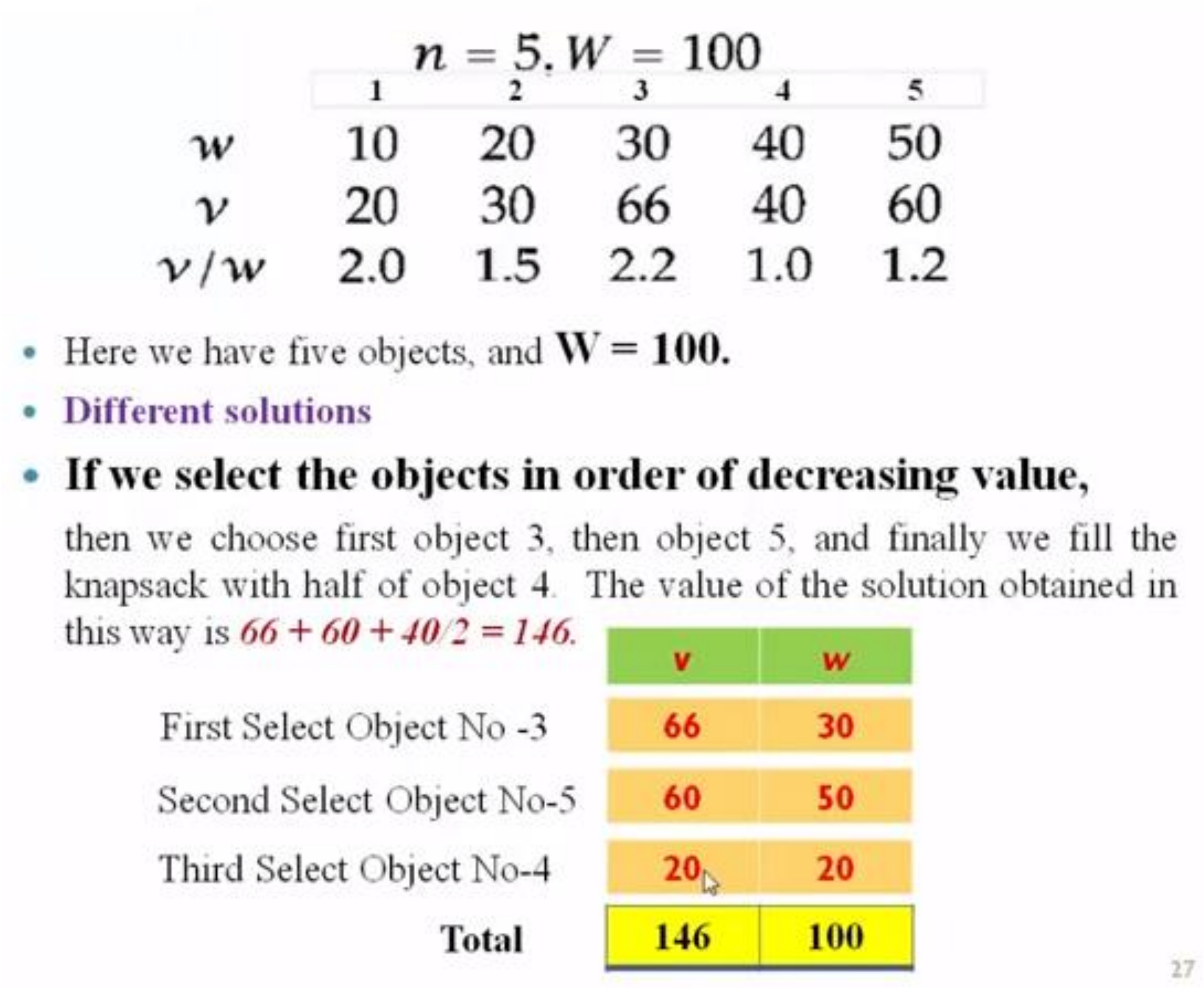
If we select the objects in order of decreasing value.

If we select the objects in order of increasing weight

**As per Grredy Method :**

**If we select the objects in order of decreasing vi /wi,**

# Fractional Knapsack problem



$n = 5, W = 100$

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $w$ | 10 | 20 | 30 | 40 | 50 |
| $v$ | 20 | 30 | 66 | 40 | 60 |
| $v/w$ | 2.0 | 1.5 | 2.2 | 1.0 | 1.2 |

- Here we have five objects, and **W = 100.**
- **Different solutions**
- **If we select the objects in order of decreasing value,**

then we choose first object 3, then object 5, and finally we fill the knapsack with half of object 4. The value of the solution obtained in this way is *66 + 60 + 40/2 = 146.*

| | v | w |
|---|---|---|
| First Select Object No -3 | 66 | 30 |
| Second Select Object No-5 | 60 | 50 |
| Third Select Object No-4 | 20 | 20 |
| **Total** | **146** | **100** |

27

## Fractional Knapsack problem

- **If we select the objects in order of increasing weight,** then we choose objects 1, 2, 3 and 4 in that order, and now the knapsack is full. The value of this solution is *20 + 30 + 66 + 40 = 156*.

|  | v | w |
|---|---|---|
| First Select Object No -1 | 20 | 10 |
| Second Select Object No-2 | 30 | 20 |
| Third Select Object No-3 | 66 | 30 |
| Third Select Object No-4 | 40 | 40 |
| **Total** | **156** | **100** |

# Fractional Knapsack problem

- **Finally if we select the objects as per the Greedy Algorithm means select objects in order of decreasing vi /wi,** we choose first object 3, then object 1, next object 2, and finally we fill the knapsack with four-fifths of object 5. Using this tactic, the value of the solution is *20 + 30 + 66 + 0.8 x 60 =*

$n = 5, W = 100$

|  | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $w$ | 10 | 20 | 30 | 40 | 50 |
| $v$ | 20 | 30 | 66 | 40 | 60 |
| $v/w$ | 2.0 | 1.5 | 2.2 | 1.0 | 1.2 |

|  | v | w |
|---|---|---|
| First Select Object No -3 | 66 | 30 |
| Second Select Object No-1 | 20 | 10 |
| Third Select Object No-2 | 30 | 20 |
| Third Select Object No-5 | 60 | 50 |
| **Total** | 170 | 110 |

## Fractional Knapsack problem

- **Finally if we select the objects as per the Greedy Algorithm means select objects in order of decreasing vi /wi,** we choose first object 3, then object 1, next object 2, and finally we fill the knapsack with four-fifths of object 5. Using this tactic, the value of the solution is *20 + 30 + 66 + 0.8 x 60 =*

$n = 5, W = 100$

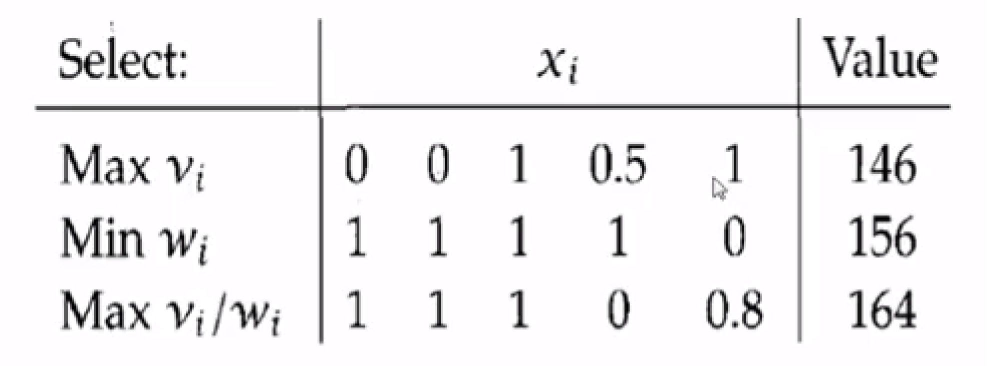
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| $w$ | 10 | 20 | 30 | 40 | 50 |
| $v$ | 20 | 30 | 66 | 40 | 60 |
| $v/w$ | 2.0 | 1.5 | 2.2 | 1.0 | 1.2 |

| | v | w |
|---|---|---|
| First Select Object No -3 | 66 | 30 |
| Second Select Object No-1 | 20 | 10 |
| Third Select Object No-2 | 30 | 20 |
| Third Select Object No-5 | 48 | 40 |
| **Total** | **164** | **100** |

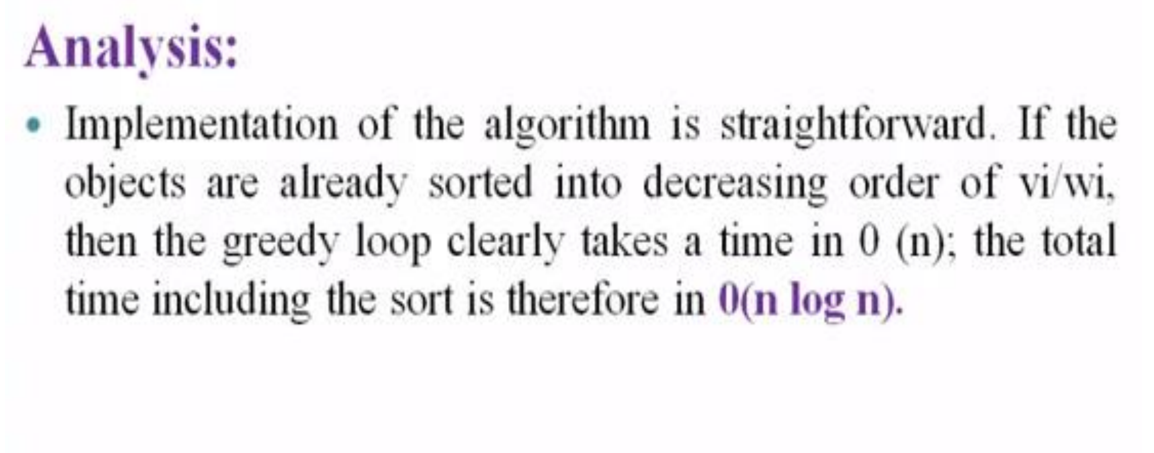
# Fractional Knapsack problem

**Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)**

```
for i = 1 to n
    do x[i] = 0
weight = 0
for i = 1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W - weight) / w[i]
        weight = W
        break
return x
```



| Select: | $x_i$ | | | | | Value |
|---|---|---|---|---|---|---|
| Max $v_i$ | 0 | 0 | 1 | 0.5 | 1 | 146 |
| Min $w_i$ | 1 | 1 | 1 | 1 | 0 | 156 |
| Max $v_i/w_i$ | 1 | 1 | 1 | 0 | 0.8 | 164 |



**Analysis:**

- Implementation of the algorithm is straightforward. If the objects are already sorted into decreasing order of vi/wi, then the greedy loop clearly takes a time in 0 (n); the total time including the sort is therefore in **0(n log n).**

## Job sequencing with deadlines

**The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.**

Here-

- You are given a set of jobs.
- Each job has a defined deadline and some profit associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs.
- Processor takes one unit of time to complete a job.

**<u>Approach to Solution-</u>**

- A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
- Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
- An optimal solution of the problem would be a feasible solution which gives the maximum profit.

# Job sequencing with deadlines

**Job Sequencing with Deadlines Solution using Greedy Algorithm**

1. Sort the jobs for their profit in descending order.
2. Choose the uncompleted job with high profit (i.e. first job in the array, since the array is sorted). Because it is not necessary to complete the job on the very first date, we will do/complete the job on the last day of the deadline (i.e. **Add the job to a new array/list at the index equal to its deadline day**).
3. Now again choose the next uncompleted high-profit job. Check its deadline. If its deadline is greater than the deadline of the previous chosen job, then we will do/complete this job on the last day of the deadline only if the day is empty, else will do the job on the previous empty day which is nearest to the deadline ( i.e. **Add the job to new array/list at the index equal or nearest to its deadline day, only if the array is empty at particular index**).
4. If you don't have the empty day before the job deadline (i.e. all the array index is occupied before the index=job.deadline) then do not do the job.
5. Repeat the process for every job.
6. The final array or list will give the best jobs to do for max profit.

## Job sequencing with deadlines

**PRACTICE PROBLEM BASED ON JOB SEQUENCING WITH DEADLINES-**

**Problem-**

Given the jobs, their deadlines and associated profits as shown-

| Jobs | J1 | J2 | J3 | J4 | J5 | J6 |
|---|---|---|---|---|---|---|
| Deadlines | 5 | 3 | 3 | 2 | 4 | 2 |
| Profits | 200 | 180 | 190 | 300 | 120 | 100 |

**Answer the following questions-**

1. Write the optimal schedule that gives maximum profit.
2. Are all the jobs completed in the optimal schedule?
3. What is the maximum earned profit?

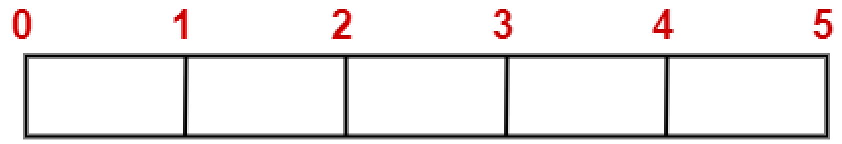
# Job sequencing with deadlines

**Solution-**

**Step-01:**
Sort all the given jobs in decreasing order of their profit-

| Jobs | J4 | J1 | J3 | J2 | J5 | J6 |
|---|---|---|---|---|---|---|
| **Deadlines** | 2 | 5 | 3 | 3 | 4 | 2 |
| **Profits** | 300 | 200 | 190 | 180 | 120 | 100 |

**Step-02:**
Value of maximum deadline = 5.
So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



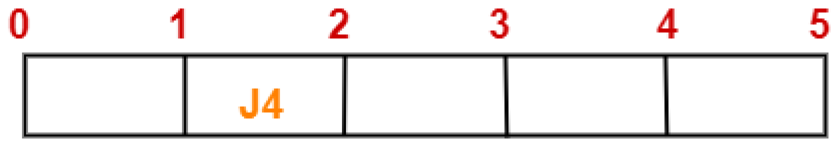
Now,

- We take each job one by one in the order they appear in **Step-01.**
- We place the job on chart as far as possible from 0.

## Job sequencing with deadlines

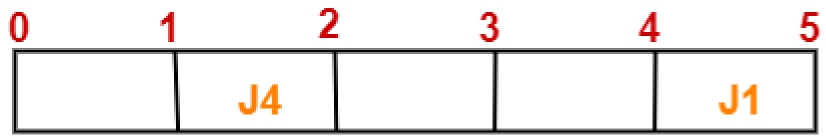
**Step-03:**

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



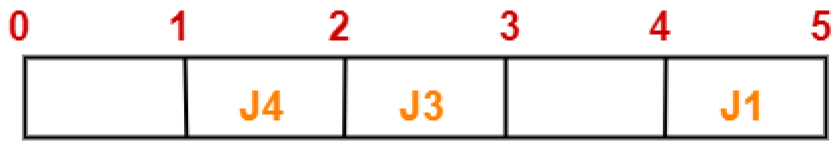
**Step-04:**

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



**Step-05:**

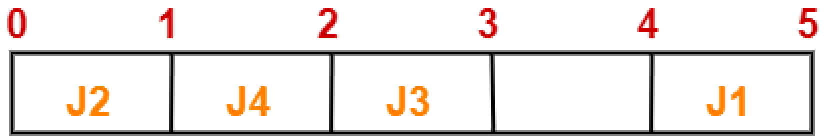
- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



## Job sequencing with deadlines

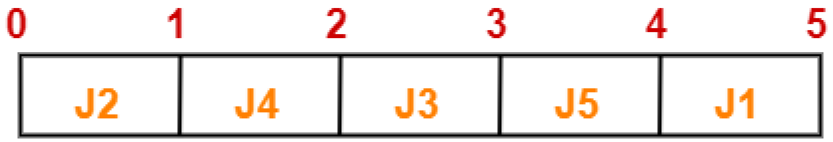
**Step-06:**

- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-



**Step-07:**

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 can not be completed.

# Job sequencing with deadlines

**Now, the given questions may be answered as-**

**Part-01:**

The optimal schedule is- **J2 , J4 , J3 , J5 , J1**

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

**Part-02:**

All the jobs are not completed in optimal schedule.

This is because job J6 could not be completed within its deadline.

**Part-03:**

Maximum earned profit

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 +Profit of job J1

= 180 + 300 + 190 + 120 + 200

= 990 units