# Greedy algorithms

- ## Prim MST Algorithm

Kruskal MST

Fractional Knapsack

# Greedy Algorithms

Prim MST

Dijkstra shortest path

# Control abstraction for Greedy Algorithm

```
Algorithm Greedy (A:set; n:integer){
   MakeEmpty(solution);
   for(i= 2;i <=n;i+ +){
   x=Select(A);
   if Feasible(solution, x) then
   solution=Union(solution;{x})
   }
   return solution
}
```

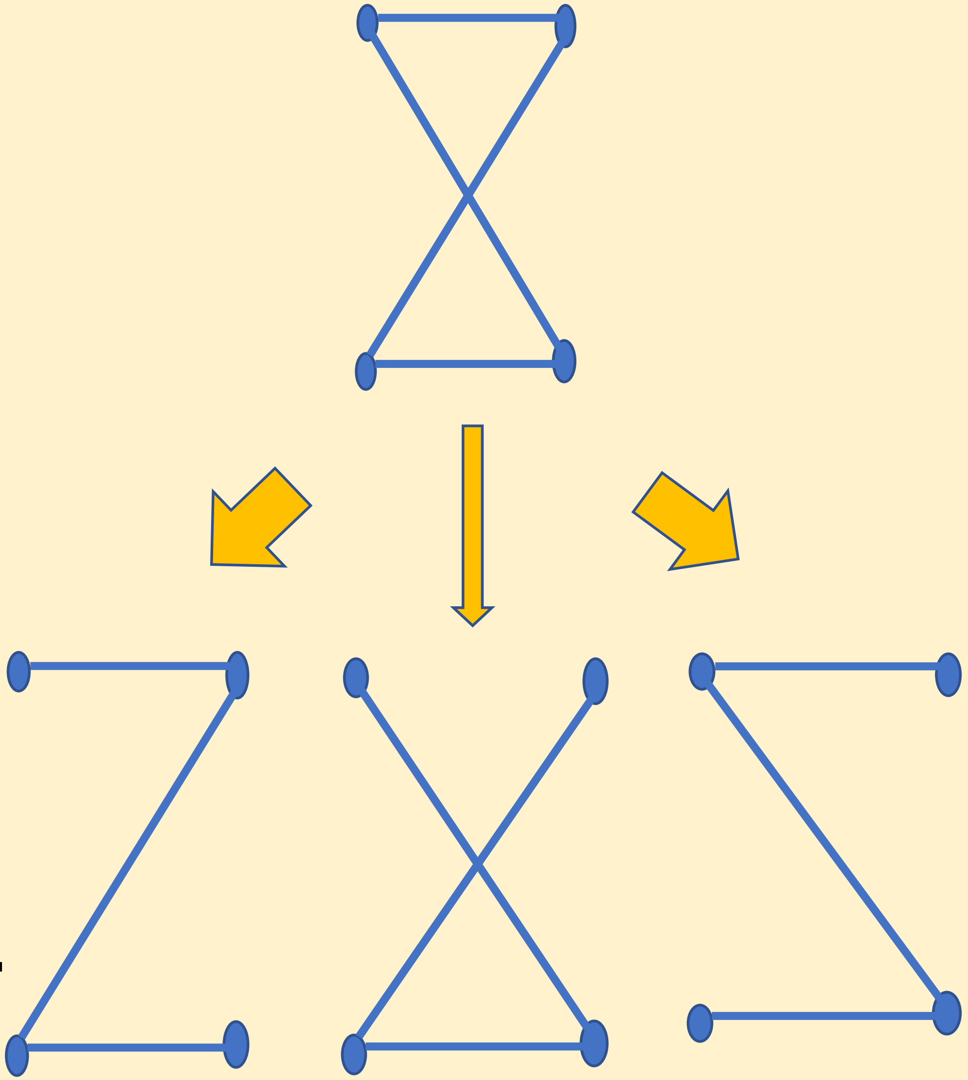
- The function *Greedy* describes the essential way that a greedy algorithm will look, once a particular problem is chosen functions *Select, Feasible, and Union* are properly implemented.

- The function *Select* selects an input from *A* whose value is assign to *x*.
- *Feasible* is a Boolean-valued function that determines if *x* can be included into the solution vector.
- The function *Union* combines *x* with the solution, and update the objective function.

```
Algorithm Greedy (A:set; n:integer){
   MakeEmpty(solution);
   for(i= 2;i <=n;i+ +){
       x=Select(A);
      if Feasible(solution, x) then
      solution=Union(solution;{x})
      }
      return solution
}
```

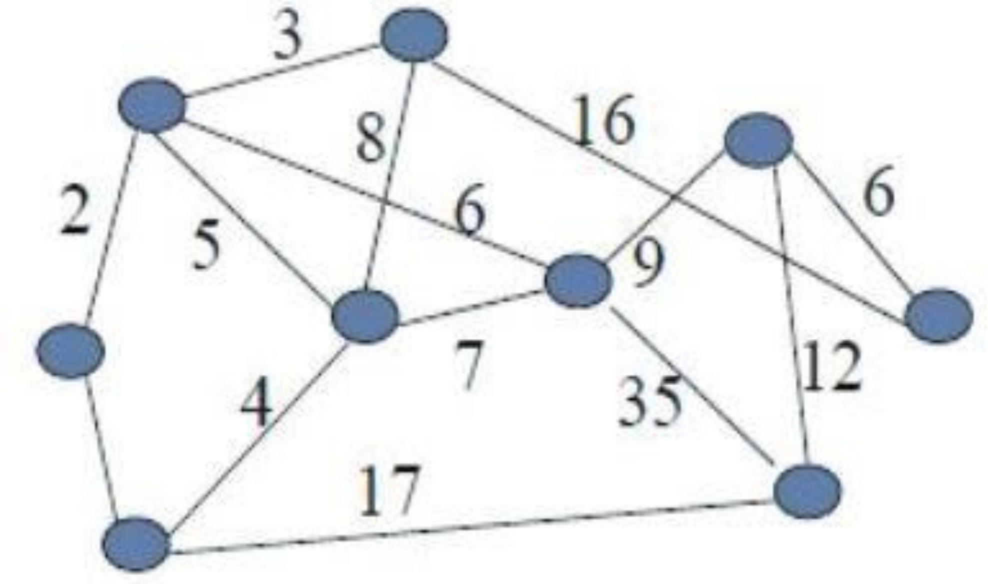
## Spanning tree

- Let $G = (V, E)$ be an undirected connected graph.
- A subgraph $T = (V, E')$ of G is a spanning tree of $G$ if $T$ is a tree.
  e.g. three spanning trees of $G$.



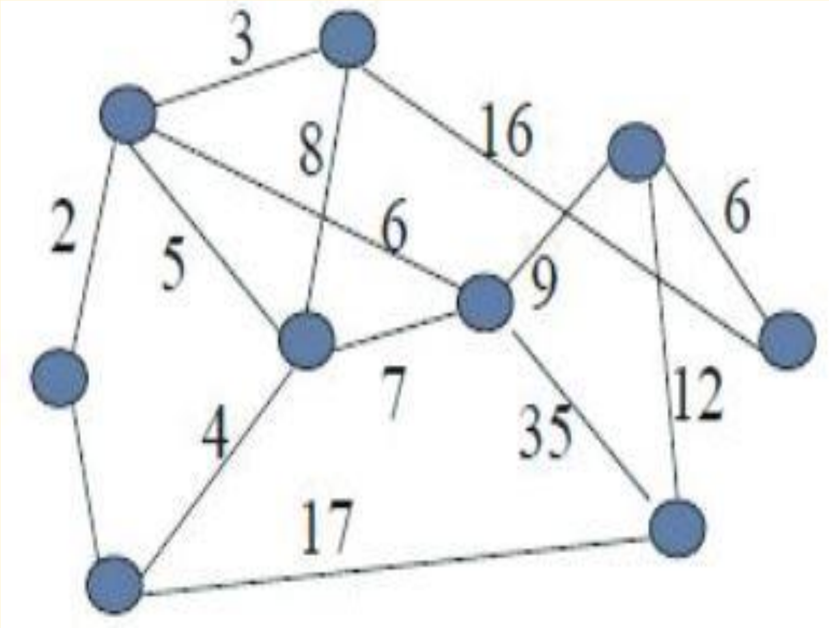
- Both $T$ and $G$ have same set of vertices $V$.
- $T$ is connected but has no cycles.
- $T$ has $(n-1)$ edges, $n$ is the number of vertices.

## Minimum Spanning Tree (MST)




- Given a connected weighted graph $G$, wish to find a set of edges with minimum (sum) weights with all vertices connected.
- Feasible solution – The spanning trees $T=(V, E)$ represent feasible choices
- Optimal solution – The spanning tree $T_{opt} =(V, E)$ with the lowest total edge cost.
- In practical applications, the edges have weights assigned to them, which may represent some cost, e. g. the length of link, etc

# MST applications

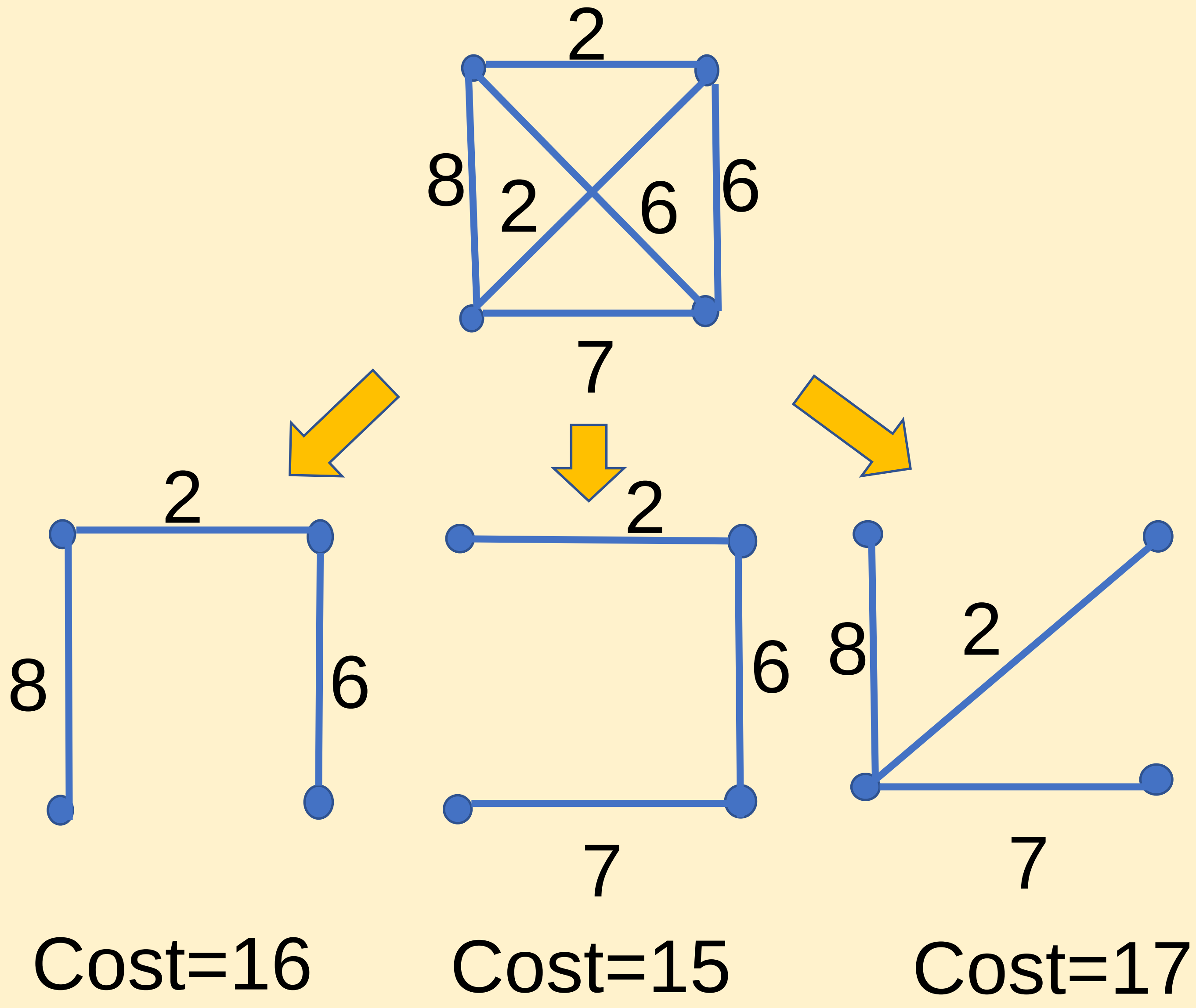



- MST a fundamental problem that arises in many scenarios

- Network design ( Telephone, electric, computer, road, …)
- Image processing, LDPC codes
- Protocol in networks (Multicast, auto-config protocol for Ethernet bridges )
- Cluster Analysis (Image, data, ...)
- Approximate algorithms for NP-hard problems (TSP)

# MST example

- What is MST?
- How many more trees can you find ?
- What is the smallest cost?
- Can you find a smaller cost spanning tree?



## MST algorithms

- Brute Force

  For small graphs, MST by enumeration may be possible.
- Since the identification of a MST involves the selection of a subset of edges, fits the subset paradigm.
- We will look at two Greedy Algorithms.
  - Kruskal greedily expand a forest of trees into an MST
  - Prim greedily grow a tree from a starting vertex
- Both yield an optimal solution.

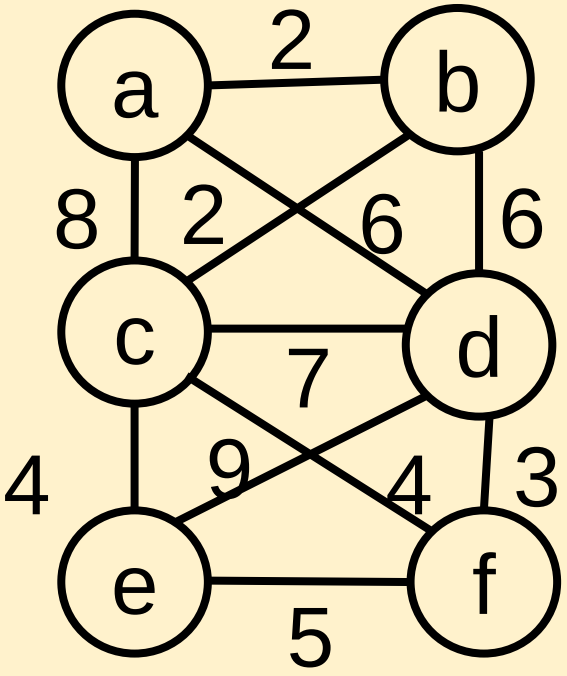
**Prim’s greedy Strategy**

- Start with a single node tree by arbitrarily choosing a root node.
- At each irrevocable step expand the current tree in a greedy manner by simply attaching a nearest node not in the tree.
- Of all possible nearest nodes choose the one which minimizes the attachment cost.
- Include this edge into the current tree.
- Continue until $(n-1)$ nodes have been explored.

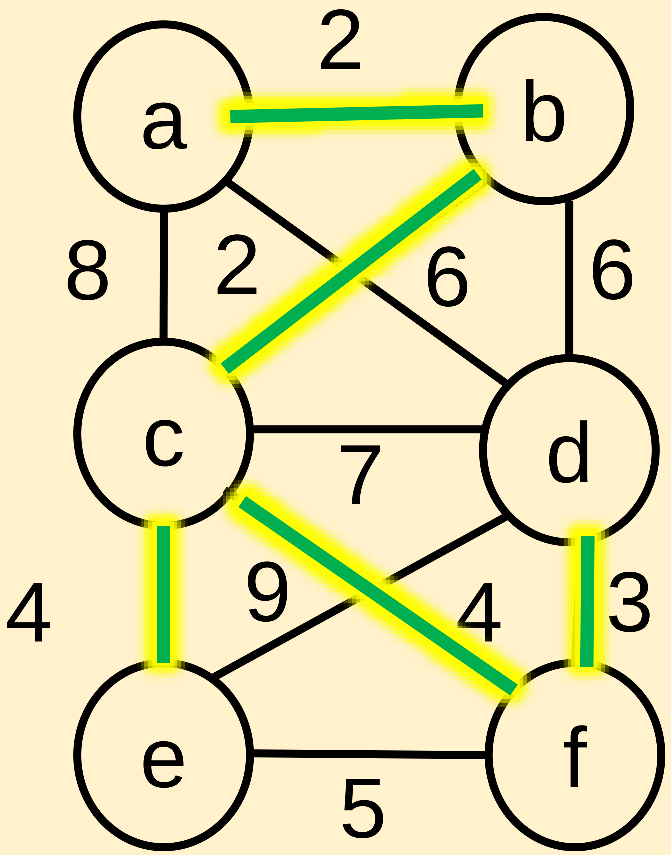
## Example: MST communication network

- Given a set of cities and communication links between them and a cost associated with each communication link.

- Network Structure: Any connected graph with n vertices must have at least $(n-1)$ edges.
- All connected graphs with $(n-1)$ edges are ‘trees’
- The problem to is find a minimum cost network that connects all the cities.



# Example: MST communication network



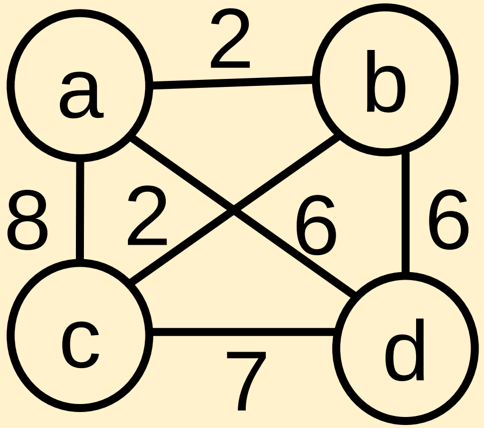

- The problem to is find a minimum cost network that connects all the cities.
- The minimum number of links needed to connect $n$ cities is $(n-1)$ .
- Start with a root node and 'greedily' grow a tree
- (a,b) Cost =2
- (a,b),(b,c) Cost =2+2=4
- (a,b),(b,c),(c,e) Cost =4+4=8
- (a,b),(b,c),(c,e),(c,f) Cost = 8+4=12
- (a,b),(b,c),(c,e),(c,f),(d,f) Cost =12+3=15

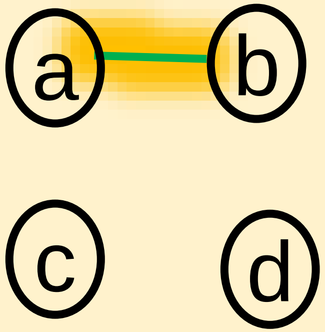
## Prim's Algorithm -- Outline

- *Input: Weighted graph of size n*
- *Output: T= (V, E). the minimum spanning tree*
- *Starting with initial vertex v0, Empty edge set,*
- *Spanning*

```
Algorithm PRIM(graph G; int n){
    T={v0};
    E'={ }, for (i= 1,i < n, i+ +){
     find a minimum weight edge e*= (v*, u*)
     among all  edges  (v, u)
    such that v is in T  and u is not in T;
     T=T ∪{u*};
     E'=E' ∪{e*};
   }
 }
```

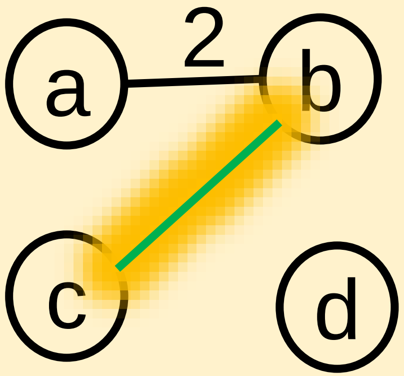
**Example:**



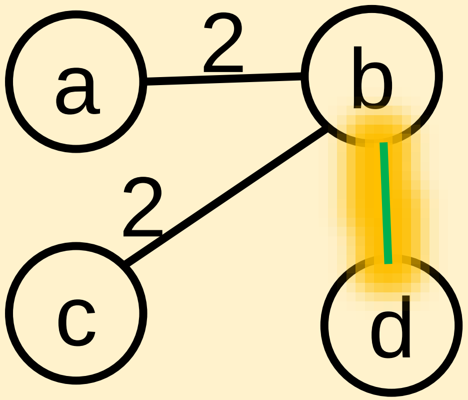




Starting from vertex **a**

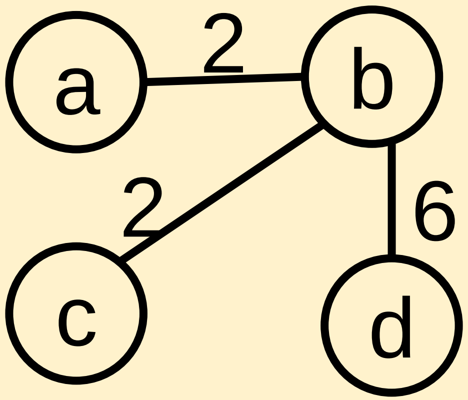
(a,b) 2
(a,c) 8
(a,d) 6




(a,c) 8
(a,d) 6
(b,c) 2
(b,d) 6




(c,d) 7
(b,d) 6
(a,d) 6



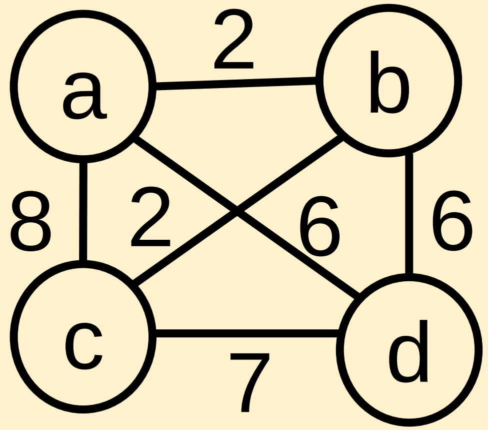

*E'=*
*{ (a,b), (b,c),(b,d) }*
*N=4, |E| =n-1=3*
*Cost =10*

## **Prim's MST -** Towards an implementation

The process of construction indicates the following:

- A data structure with the cost of edges (e.g. an adjacency matrix)

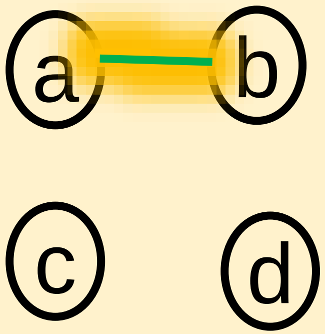
$$Cost = \begin{bmatrix} 0 & 2 & 8 & 6 \\ 2 & 0 & 2 & 6 \\ 8 & 2 & 0 & 7 \\ 6 & 6 & 7 & 0 \end{bmatrix}$$

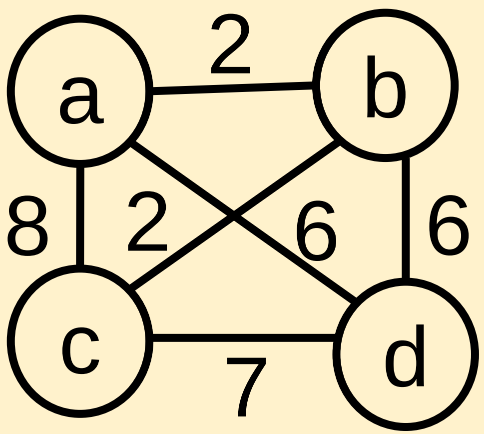



- A structure indicating for each vertex the possible edges incident on them but so far unused.
- An observation that any edge that results in moving to vertex already visited will form a cycle and is not admissible.
- The tree $T$ can be represented by the edge set $E'$ as all vertices are included.

## To find the next minimum cost edge efficiently

- Associate with every vertex *j* not in the tree a value *Next(j)* (*Next(j) = 0* for all vertices already in the tree).
- *Next(j)* is a vertex in the tree such that *cost[j, next(j)]* is minimum among all choices for *Next(j)*




*Next (b)= a*

*cost (b, Next(b))= cost (b, a) =2*

*Next (c)= a*

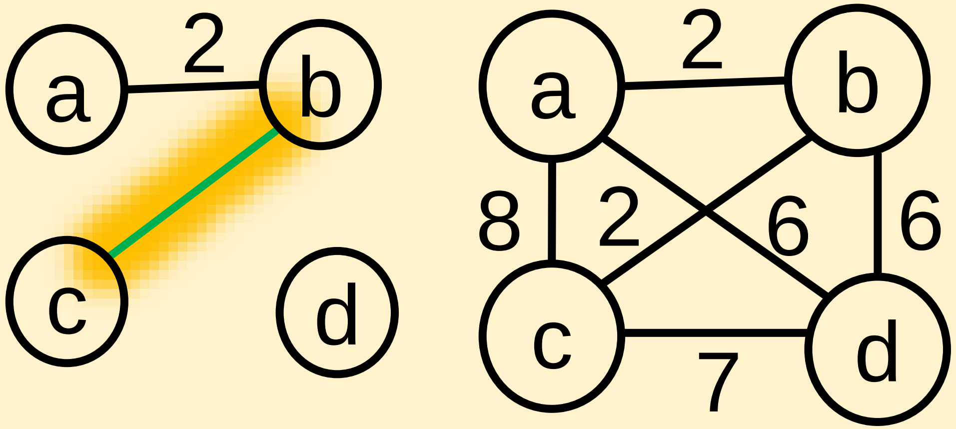
*cost (c, Next(c))= cost (c, a) =8*

*Next (d)= a*

*cost (d, Next(d))= cost (d, a) =6*

## To find the next minimum cost edge efficiently

- Associate with every vertex *j* not in the tree a value *Next(j)* (*Next(j) = 0* for all vertices already in the tree).
- *Next(j)* is a vertex in the tree such that *cost[j, next(j)]* is minimum among all choices for *Next(j)*




*Next (c)= b (NOT a)*

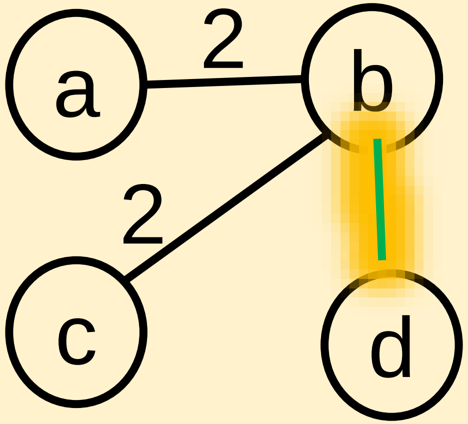
*cost (c, Next(c))= cost (c, b) =2*

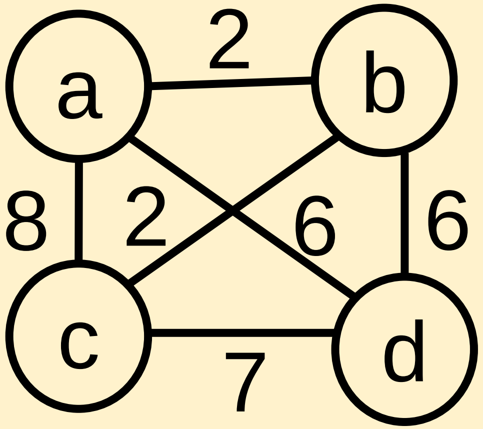
*Next (d)= a (OR b)*

*cost (d, Next(d))= cost (d, a) =6*

## To find the next minimum cost edge efficiently

- Associate with every vertex *j* not in the tree a value *Next(j)* (*Next(j) = 0* for all vertices already in the tree).
- *Next(j)* is a vertex in the tree such that *cost[j, next(j)]* is minimum among all choices for *Next(j)*




*Next (d)= a (OR b, NOT c )*
*cost (d, Next(d))= cost (d, a) =6*

# Prim's Algorithm – An implementation

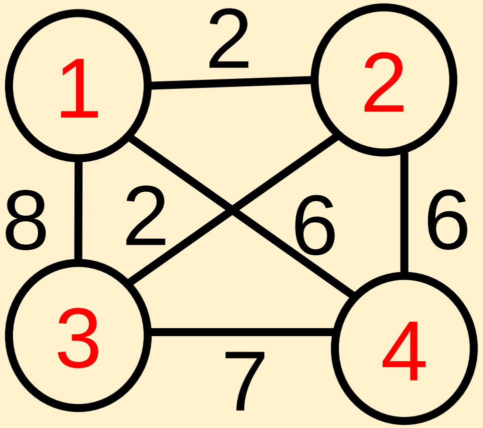
- *E and T are implemented as 1−D arrays of pairs(i;j)*
- *Cost is an n by n adjacency matrix for G*
- *E the set of edge in G*
- *Next: array[1…n]of integer*
- *Start spanning*
- *find which edges are possible from vertex k or m*
- *Don't reuse k or m*

```
Algorithm PRIM(E; T:Set; Cost: Matrix; n;
mincost:integer){
    (k, m) =position of the minimum cost edge in Cost;
    mincost=Cost[k, m];
   T[1] = (k, m);
   for(i= 1,i <=n, i+ +){
      if(Cost(i, m)> Cost(i,k))
         Next[i] =k
         else Next[i] =m}
         Next[k] = 0, Next[m] = 0
         for(i= 2;i <=n−1;i+ +){
          j=value such that Next[j] != 0 and Cost[j, Next[j]]
      is a minimum
```

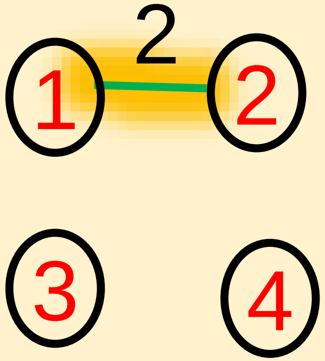
- *E and T are implemented as 1−D arrays of pairs(i;j)*
- *Cost is an n by n adjacency matrix for G*
- *E the set of edge in G*
- *Next: array[1…n]of integer*
- *Start spanning*
- *find which edges are possible from vertex k or m*
- *Don't reuse k or m*
- *For remaining edges*
- *vertex j in T*
- *Adjust Next for edges available*

```
        mincost=mincost+Cost[j, next[j]]);
        T[i] = (j, Next(j))
        Next[j] = 0
        for(k= 1;k <=n;k+ +){
        if(Next[k]! = 0) and (Cost[k, Next[k]]>
      Cost[k, j])
        Next[k] =j;}
    }
    if mincost=∞, then print('No tree possible');
}
```

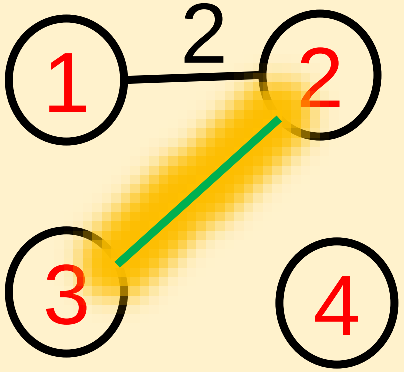
**Example:**



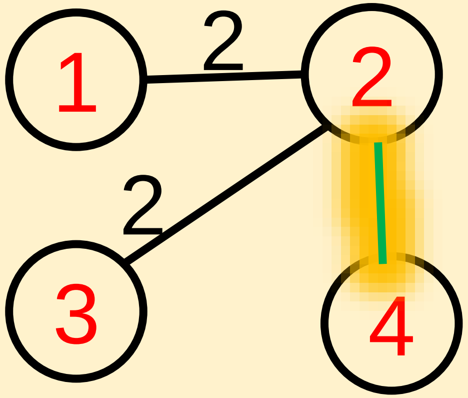

$$cost = \begin{bmatrix} 0 & 2 & 8 & 6 \\ 2 & 0 & 2 & 6 \\ 8 & 2 & 0 & 7 \\ 6 & 6 & 7 & 0 \end{bmatrix}$$



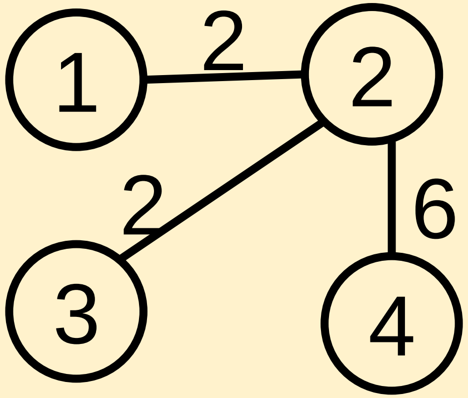

*T[1]=(1,2)*
*mincost =2*
*Next [1]=0*
*Next [2]=0*
*Next[3]=2*
*Next[4]=2*



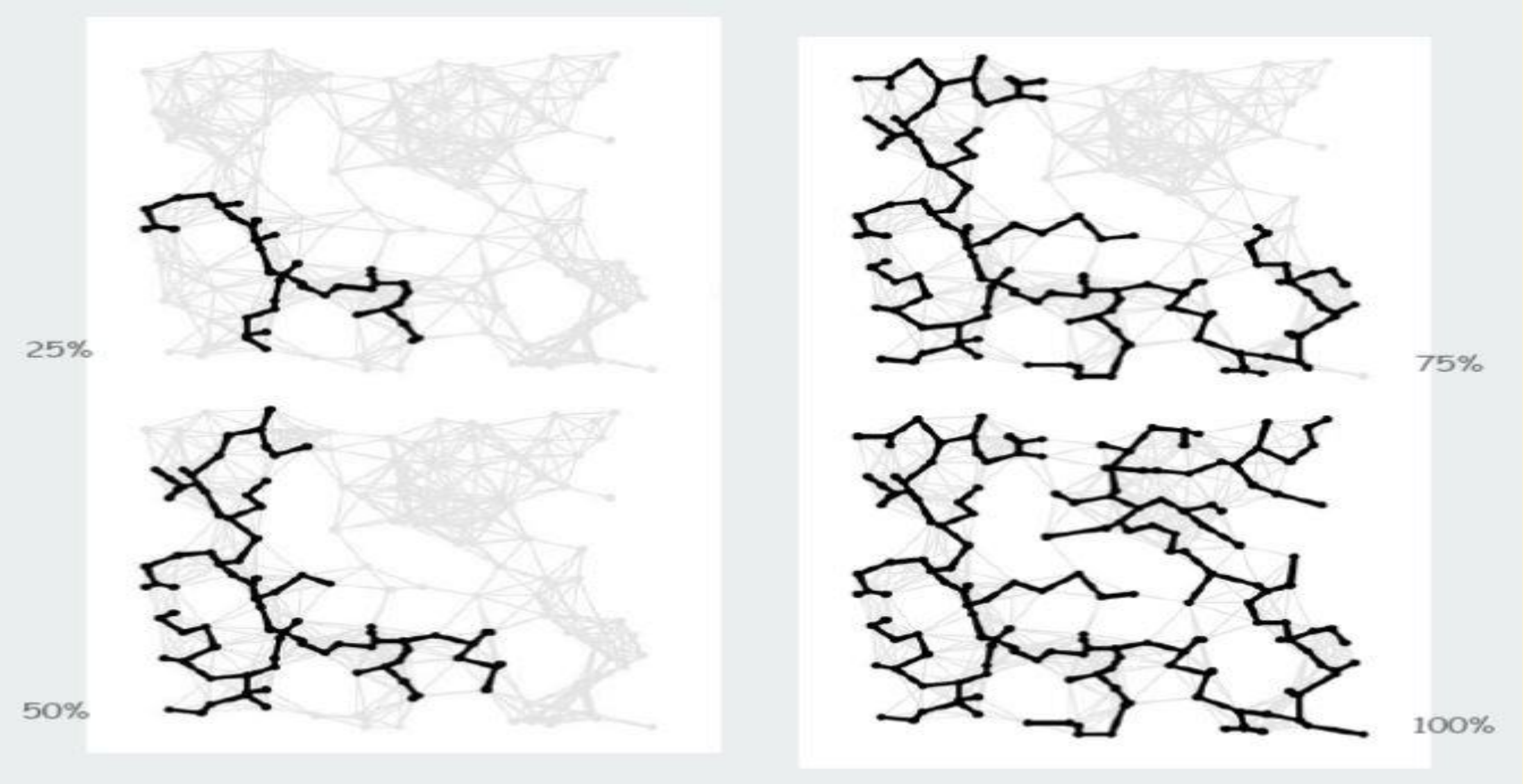

*i=2, j=3; T[2]=(3,2)*
*mincost =2+2=4*
*Next [1]=0*
*Next [2]=0*
*Next[3]=0*
*Next[4]=2*

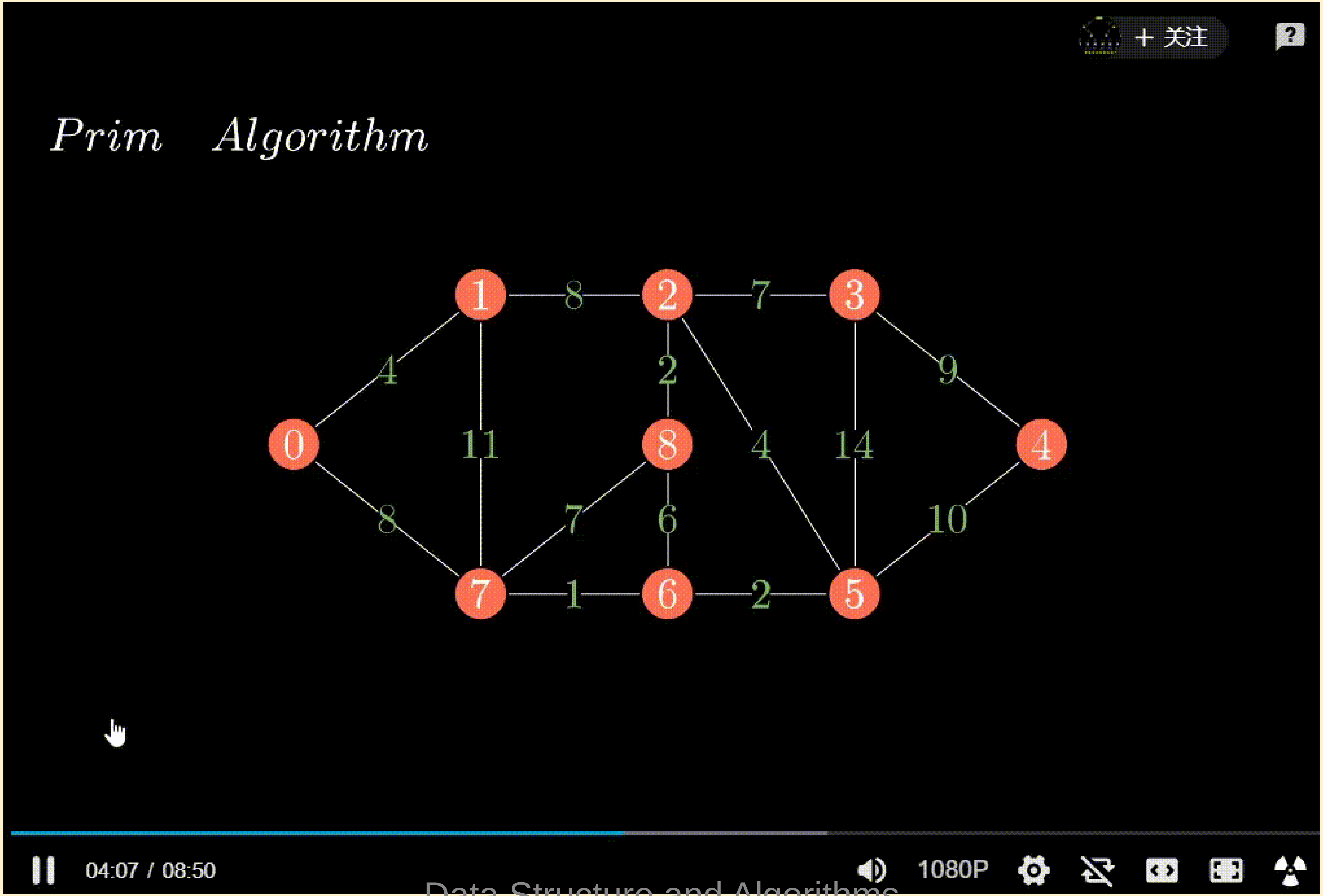


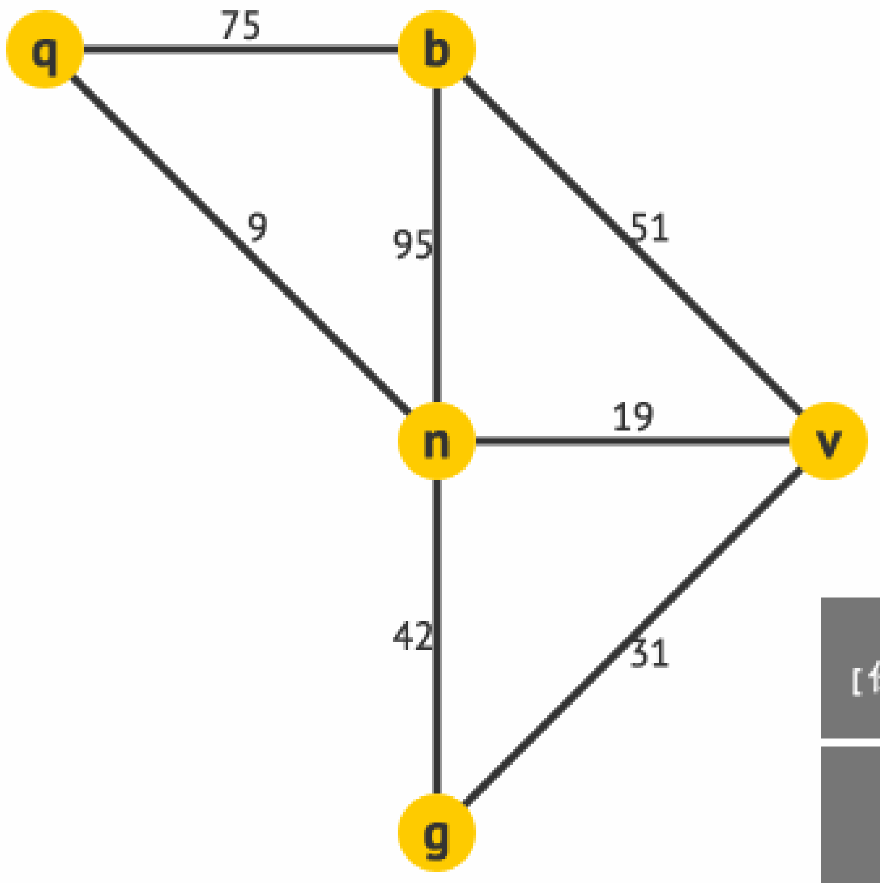

*i=2, j=4; T[3]=(4,2)*
*mincost =4+6=10*
*Next [1]=0*
*Next [2]=0*
*Next[3]=0*
*Next[4]=0*



# Prim MST growing tree illustration




+ 关注
Prim Algorithm
0
1
2
3
4
5
6
7
8
4
8
11
7
2
6
1
2
4
14
9
10
04:07 / 08:50
1080P




[任务] Prim 算法演示

# Greedy algorithms

◆ Kruskal MST

Kruskal MST

Fractional Knapsack

# Greedy Algorithms

Prim MST

Dijkstra shortest path

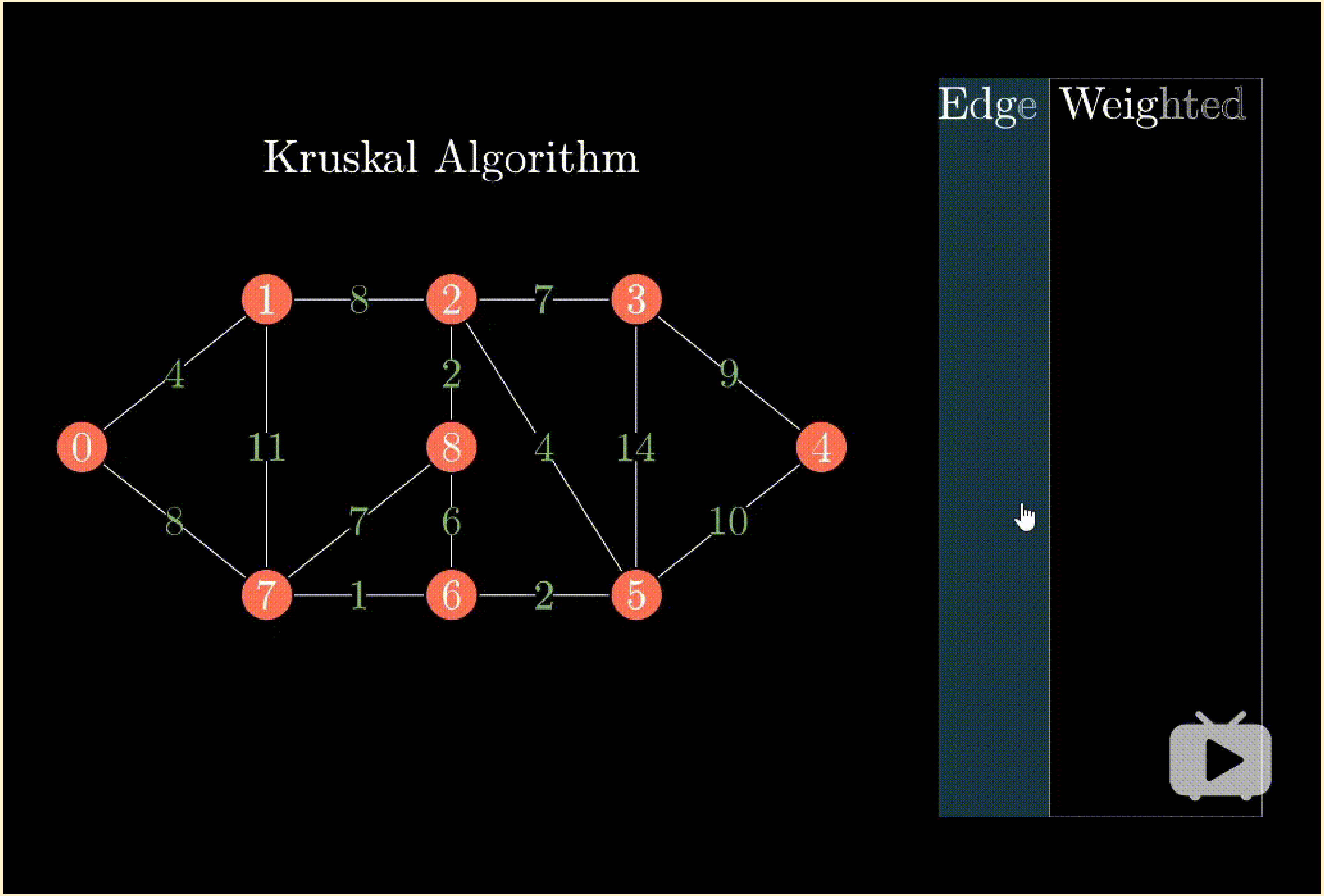
# Control abstraction for Greedy Algorithm

```
Algorithm Greedy (A:set; n:integer){
   MakeEmpty(solution);
   for(i= 2;i <=n;i+ +){
   x=Select(A);
   if Feasible(solution, x) then
   solution=Union(solution;{x})
   }
   return solution
}
```

- The function *Greedy* describes the essential way that a greedy algorithm will look, once a particular problem is chosen functions *Select, Feasible, and Union* are properly implemented.

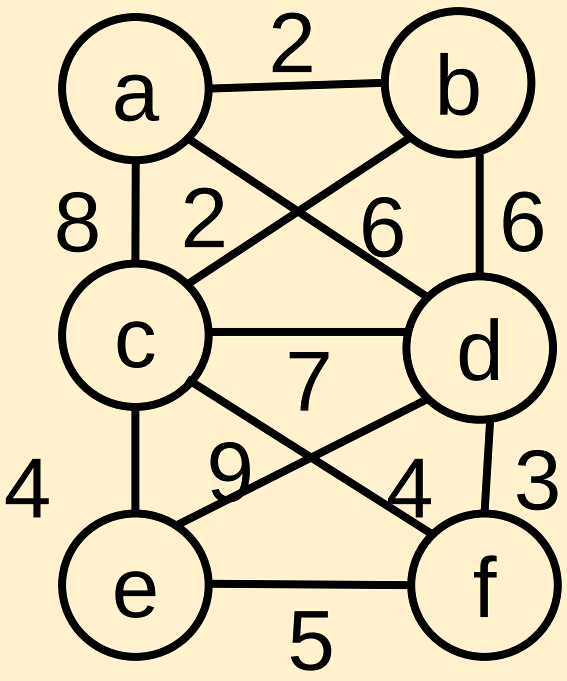
- The function *Select* selects an input from *A* whose value is assign to *x*.
- *Feasible* is a Boolean-valued function that determines if *x* can be included into the solution vector.
- The function *Union* combines *x* with the solution, and update the objective function.

```
Algorithm Greedy (A:set; n:integer){
   MakeEmpty(solution);
   for(i= 2;i <=n;i+ +){
   x=Select(A);
   if Feasible(solution, x) then
   solution=Union(solution;{x})
   }
   return solution
}
```


Kruskal Algorithm
Edge
Weighted
0
1
2
3
4
5
6
7
8
4
8
7
9
2
11
14
10
1
2
6
7

## Kruskal's Greedy Strategy:

`greedily' expand a sequence of sub-graphs into an acyclic bigger subgraph that is a tree.

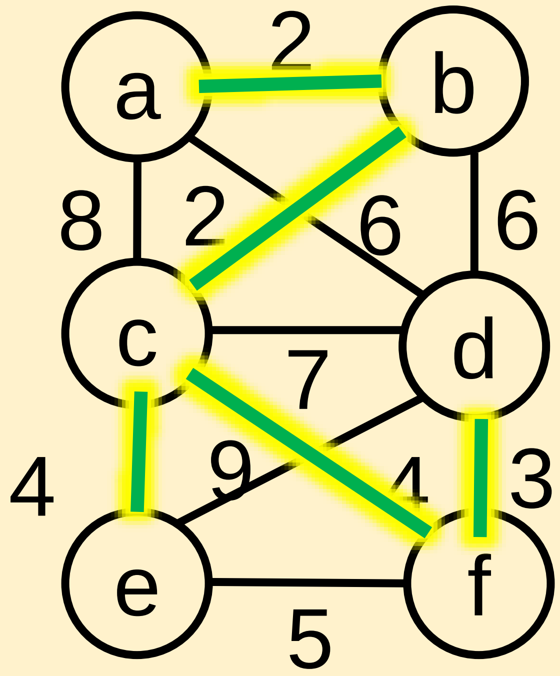



- Sort the edges in increasing order
- Start with an empty sub-graph
- Add the next edge on the list to the current graph
- If an inclusion results in a cycle discard the edge

- Sort the edges in increasing order of cost

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

- $n=6$, so $n-1 = 5$ to make $T$
- Sorting takes $O(e\log e)$ where $e = |E|$




$E_1 = (a,b),\ T = \{(a,b)\},\ cost = 2$

$E_2 = (b,c),\ T = \{(a,b),(b,c)\},\ cost = 4$

$E_3 = (d,f),\ T = \{(a,b),(b,c),(d,f)\},\ cost = 7$

$E_4 = (c,f),\ T = \{(a,b),(b,c),(d,f),(c,f)\},\ cost = 11$

$E_5 = (c,e),\ T = \{(a,b),(b,c),(d,f),(c,f),(c,e)\},\ cost = 15$

# Kruskal's Algorithm Outline

- *Input: Weighted graph of size n*
- *Output: T= (V, E) the minimum spanning tree*
- $e_{ik}$ *next edge*

```
Algorithm Kruskal(graph G; int n) {
    Sort edges of G in increasing
   order of weight
    T={};k= 0;
     while(T  has less than (n−1)
        edges) {
           k=k+ 1;
           if T ∪ {e_ik} is acyclic
           add e_ik to T;
   }
 return T
}
```

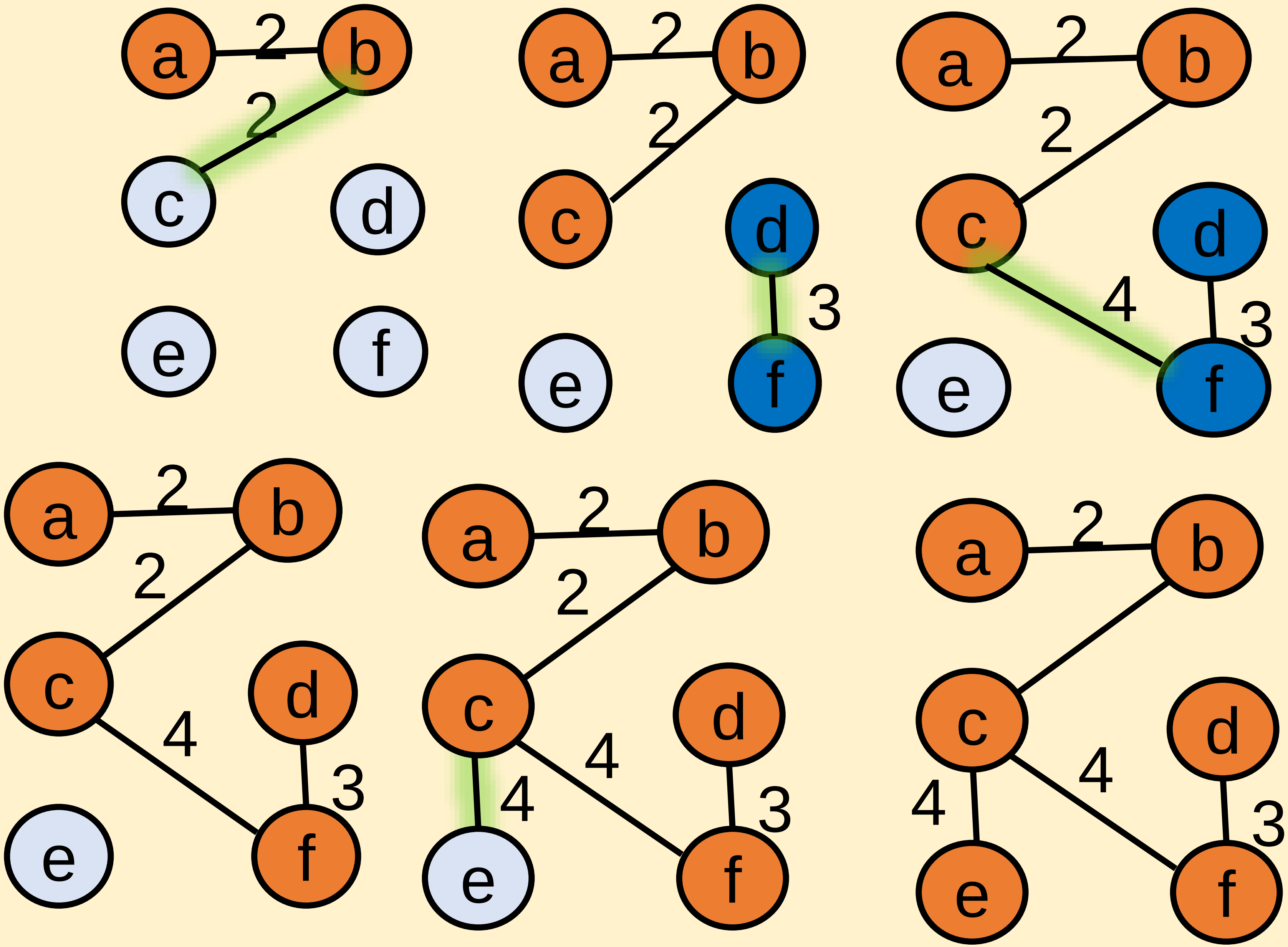
# Kruskal's Algorithm -- implementation

Main step : Cycle detection

- Check if adding an edge to $T$ creates a cycle
- Solution is connected components
- Maintain a set of connected components
- Initially each vertex in its own component
- When a new edge $(u, v)$ is selected
  - If $u$ and $v$ are in the same component, then adding vertices, $(u, v)$ would create a cycle. Thus we reject those.
  - If not a cycle, then merge the connected components containing $u$ and $v$.

- The connected components are color coded as orange and blue.

- Thus we reject vertices that are of the same color.



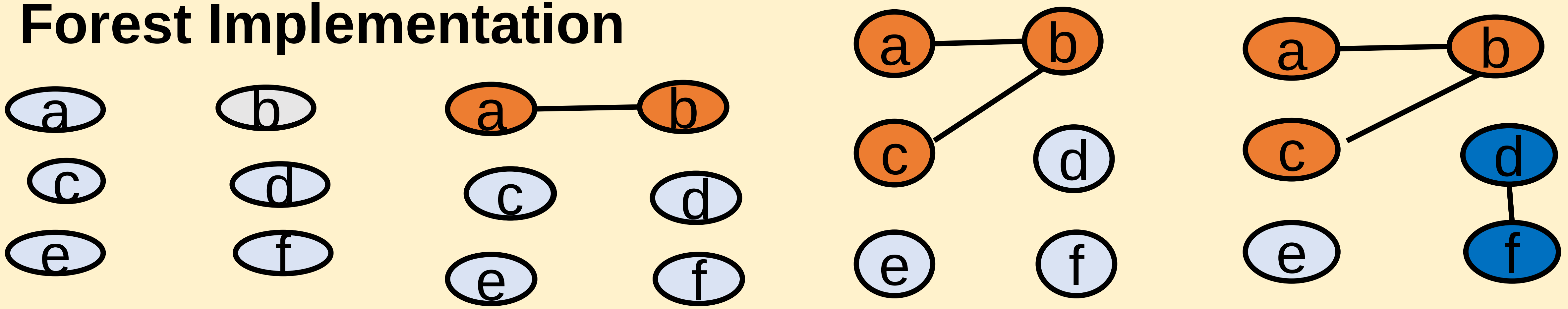
# Forest of Trees

- Use a linked list of vertices (or an array) for each connected component.

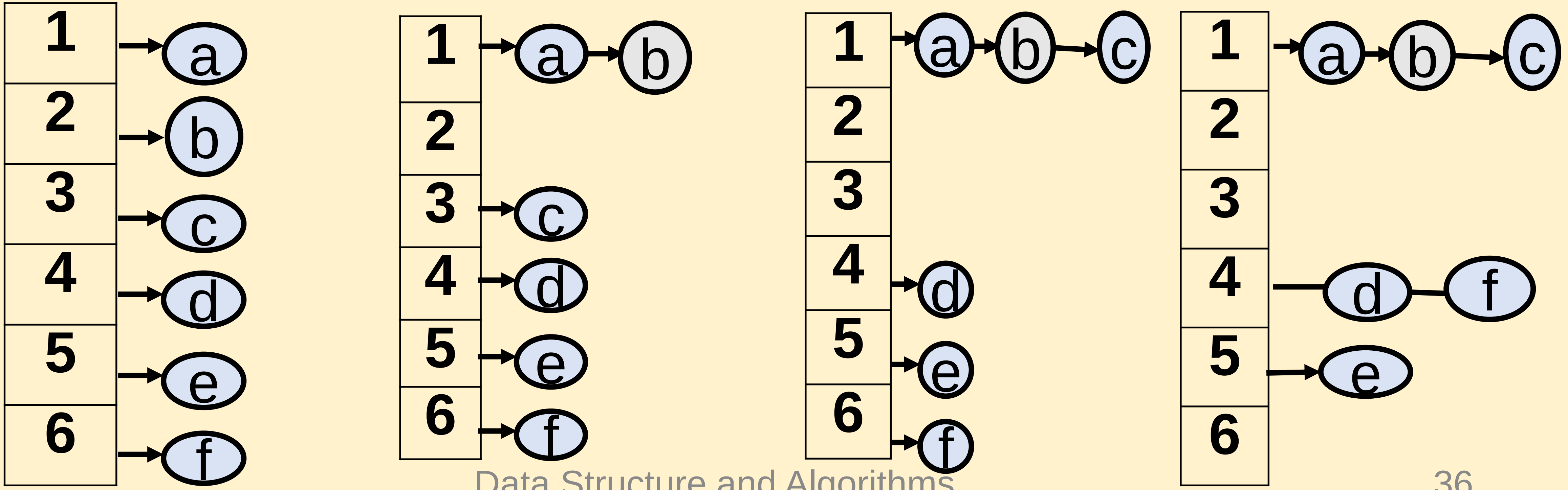
- Implement routines
  - Locate the tree contain a vertex, e.g. $FindVertex(Forest, v)$

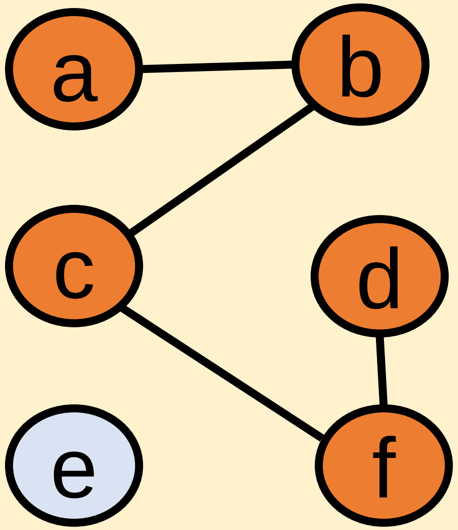
  - To merge one tree $i$ with another $j$, e.g. $Merge(Forest, i, j)$

# Forest Implementation

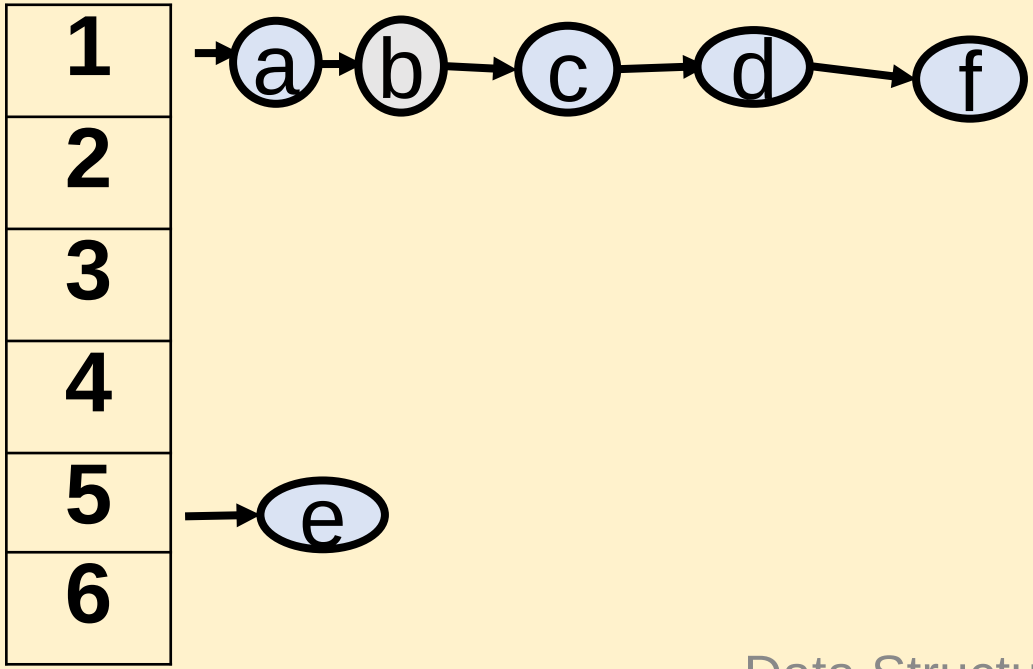


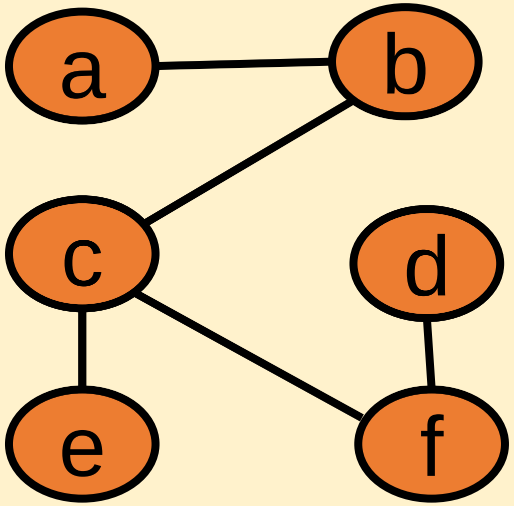

Initial forest　Merge ( forest ,1,2)　Merge ( forest ,1, 3)　Merge ( forest ,4, 6)



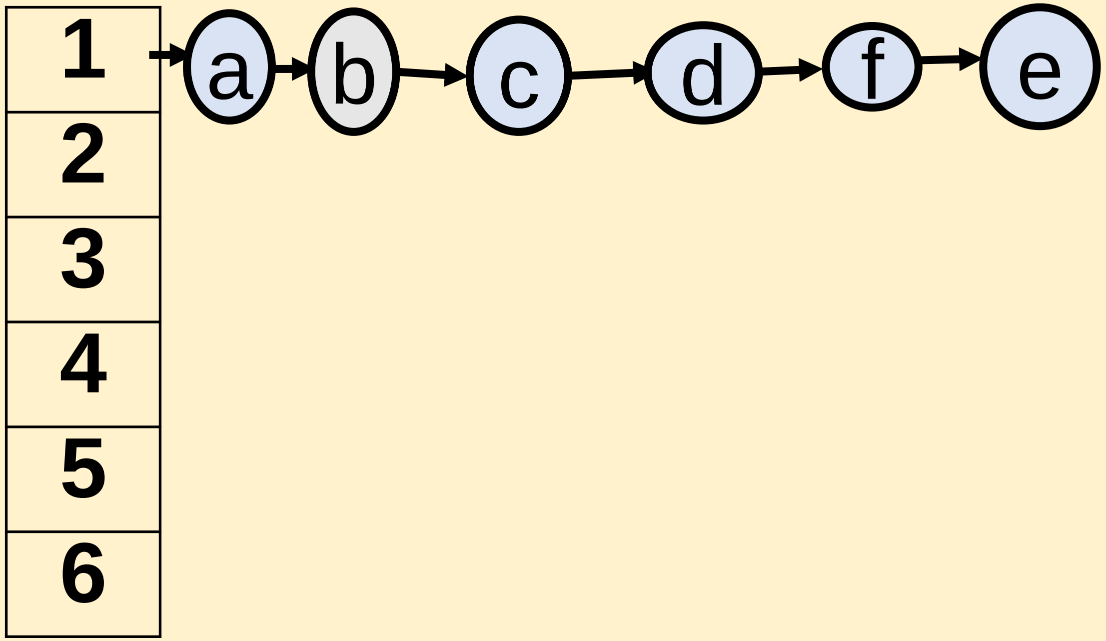



**Merge ( forest ,1,4)**




**Merge (forest ,1,5)**



# Kruskal's Algorithm –Sorted Edge List

- *Forest: Array[1::n] of Vertex List*
- *Sort edge list E in ascending order of cost*
- *Make initial forest, initial cost*
- *Current edge to add?*
- *Locate vertices in components*

```
Algorithm Kruskal(E, T :Set; Cost:matrix){
    Forest: Array[1...n]of Vertex List;
    Sort(E)
   for(i= 1;i <=n;i+ +) Insert(Forest; i);
      i= 0; mincost= 0;
        while((i <n−1)
      &(NotEmpty(Edgelist))){
       (u, v) = Next(Edgelist);
        j=FindVertex(Forest, u);
        k=FindV ertex(Forest; v) ;
```

- *Locate vertices in components*
- *If different components*
- *Add edge to spanning tree*
- *Update total cost*
- *Update forest*

```
if(j! =k){
  i=i+ 1; T[i] = (u, v);
  mincost=mincost+Cost[u, v];
  Merge(Forest, j, k)
  }
}if(i! =n−1) print('no spanning tree');
return(T, mincost);
}
```

# Kruskal's Algorithm – Heap Edge List

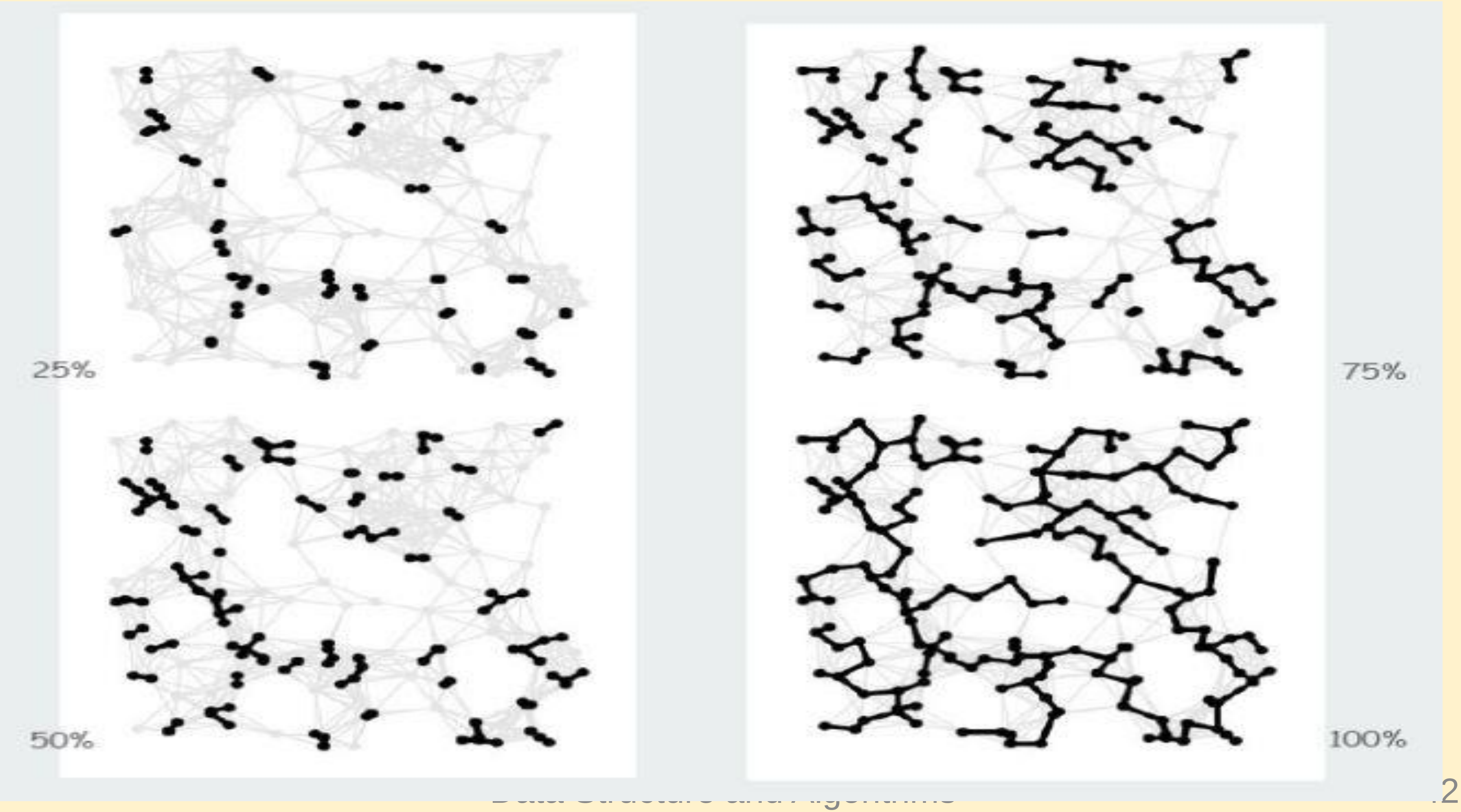
- *Forest: Array[1::n] of Vertex List*
- *Make initial forest*
- *Order edges by least cost*
- *start cost*
- *Find next min cost edge*
- *Locate vertices in components*

```
Algorithm Kruskal(E; T:Set;Cost:matrix){
  Forest: Array[1...n] of Vertex List;
   for(i= 1;i <=n;i+ +)
        Insert(Forest, i);
        MakeHeap(Edgelist, e);
         i= 0; mincost= 0;
          while((i <n−1)
        &&(NotEmpty(Edgelist))){
              (u, v) =Next(Edgelist);
              FixHeap(Edgelist, e,1)
               j=FindVertex(Forest, u);
              k=FindVertex(Forest, v)
```

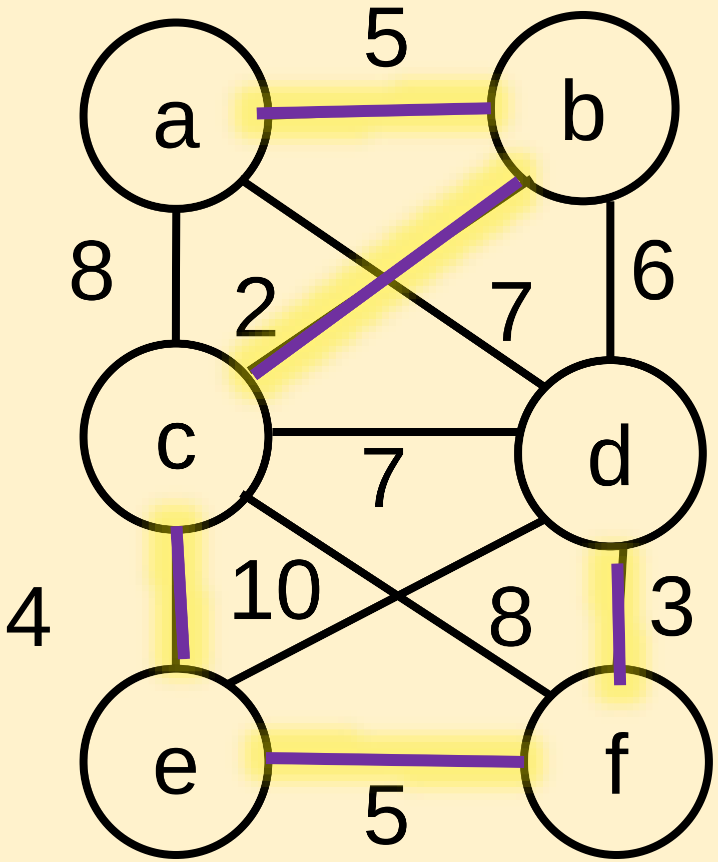
- *Locate vertices in components*
- *If different components*
- *Add edge to spanning tree*
- *Update total cost*
- *Update forest*

```
if(j! =k){
  i=i+ 1; T[i] = (u, v);
  mincost=mincost+Cost[u, v];
  Merge(Forest, j, k)
  }
}if(i! =n−1) print('no spanning tree');
return(T, mincost);
}
```

# Kruskal MST merging connected components illustration



# In class exercise:
# Find the MST using Prim algorithm



# In class exercise:
# Find the MST using Kruskal algorithm