BASIC GRAPH CONCEPTS

DEFINITION

$$G = (V, E)$$

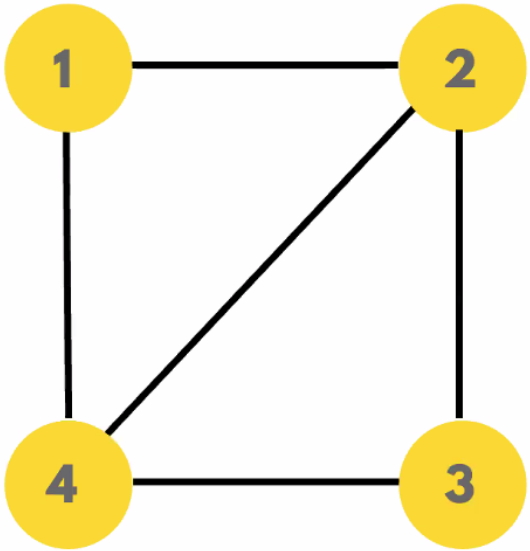
V = set of vertices

E = set of edges

$$|E| <= |V^2|$$

BASIC GRAPH CONCEPTS

DEFINITION



$$G = (V, E)$$

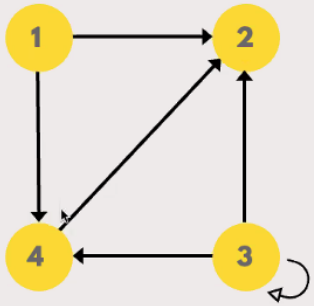
$$V = \{1,2,3,4\}$$

$$E = \{(1,2),(2,3),(2,4),(3,4),(4,1)\}$$

$$|E| <= |V^2|$$

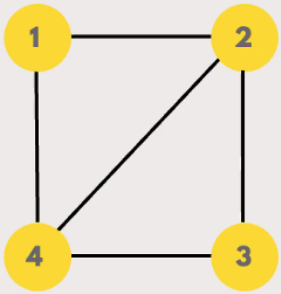
BASIC GRAPH CONCEPTS

DIRECTED VS UNDIRECTED

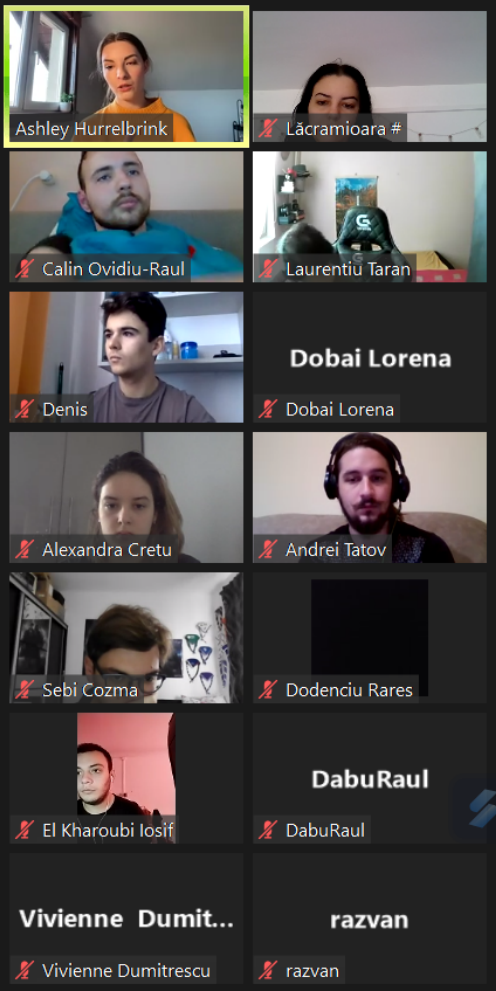


DIRECTED GRAPH

Self-loop edges are possible only in directed graphs



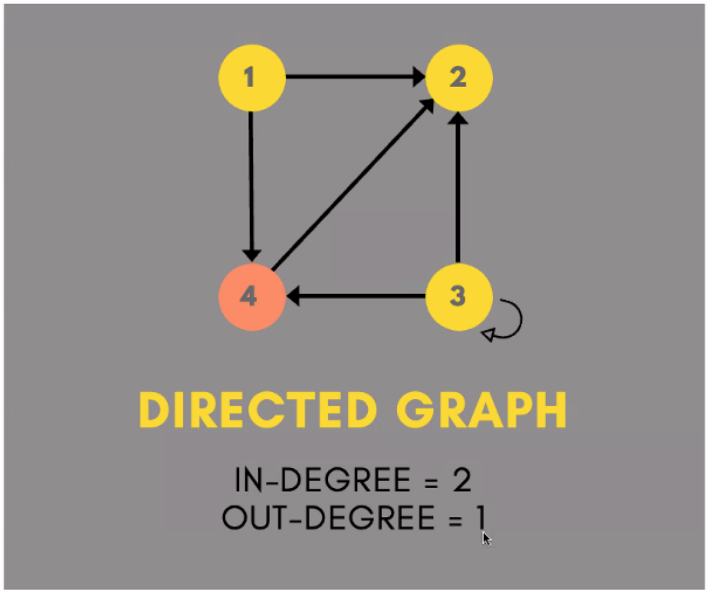
UNDIRECTED GRAPH

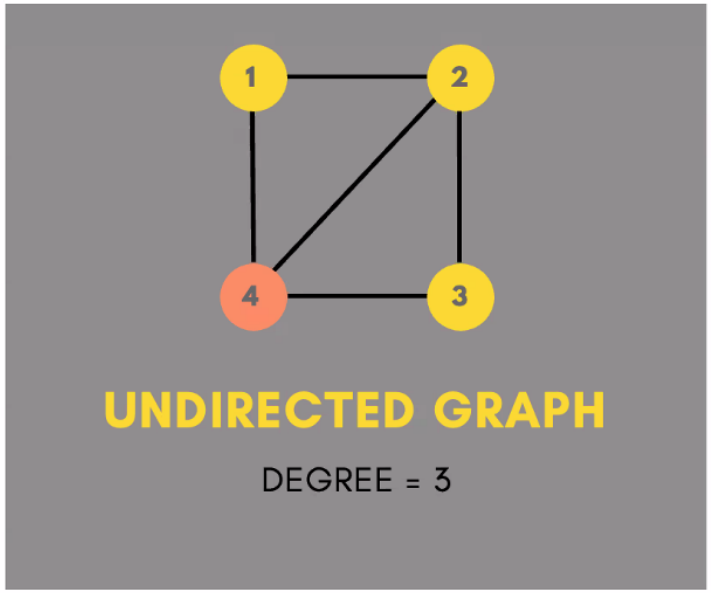


BASIC GRAPH CONCEPTS

DEGREE OF A VERTEX V

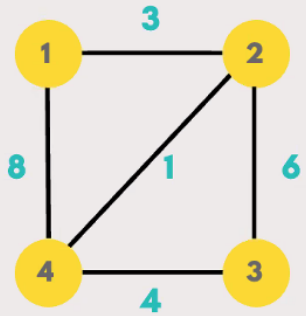
The number of edges adjacent to v





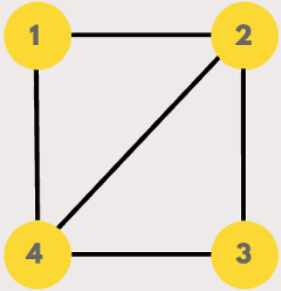
BASIC GRAPH CONCEPTS

WEIGHTED VS UNWEIGHTED



WEIGHTED GRAPH

EACH EDGE HAS AN ASSOCIATED WEIGHT (NUMERCAL VALUE)



UNWEIGHTED GRAPH

BASIC GRAPH CONCEPTS

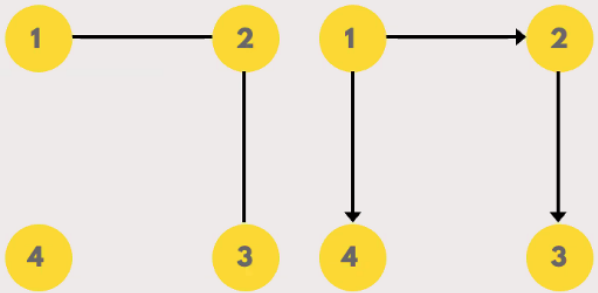
CONNECTED VS DISCONNECTED

CONNECTED

A directed graph is **STRONLGY CONNECTED** if there is a directed path between any two vertices.

An undirected graph is **CONNECTED** if there is a path between any two vertices.

DISCONNECTED



BASIC GRAPH CONCEPTS

DENSE VS SPARSE

DENSE

The number of edges is close to the maximum possible (V^2)

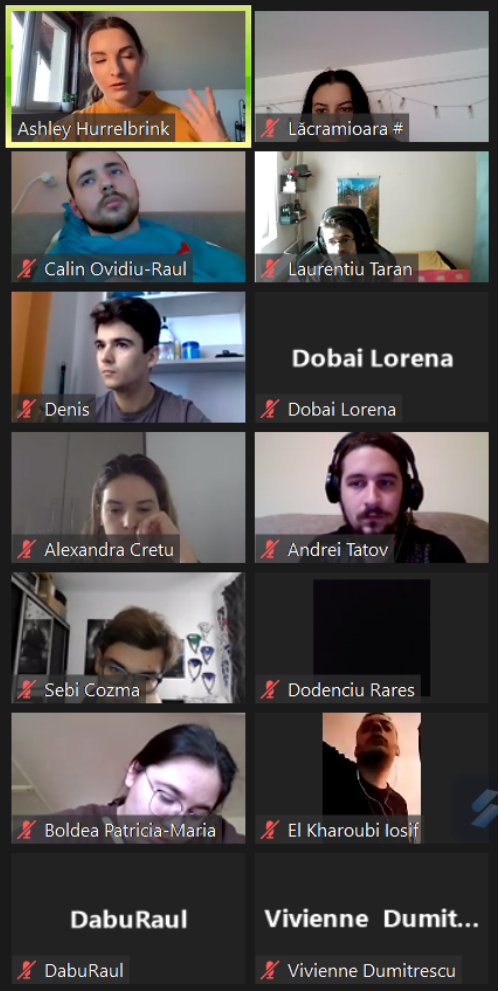
SPARSE

The number of edges is small.

GRAPH REPRESENTATION

1.ADJACENCY MATRIX

2.ADJACENCY LIST



1 GRAPH REPRESENTATION
ADJACENCY MATRIX

GRAPH REPRESENTATION

ADJACENCY MATRIX

WEIGHTED GRAPH

N*N MATRIX A

A[P, Q] = VALUE
IF THERE IS AN EDGE FROM P TO Q

A[P, Q] = 0
IF THERE IS NOT AN EDGE FROM P TO Q

UNWEIGHTED GRAPH

N*N MATRIX A

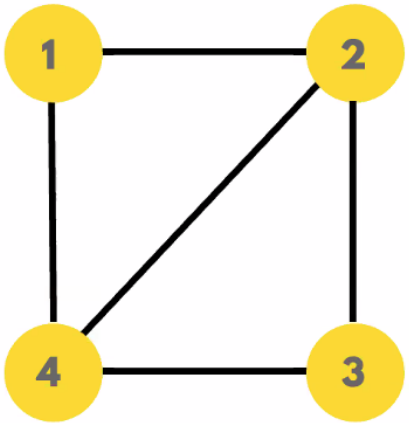
A[P, Q] = 1
IF THERE IS AN EDGE FROM P TO Q

A[P, Q] = 0
IF THERE IS NOT AN EDGE FROM P TO Q

GRAPH REPRESENTATION

ADJACENCY MATRIX

UNDIRECTED GRAPH EXAMPLE

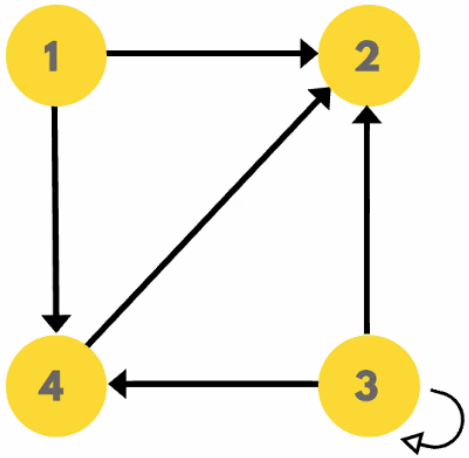


row \ column	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	1
4	1	1	1	0

GRAPH REPRESENTATION

ADJACENCY MATRIX

DIRECTED GRAPH EXAMPLE



row \ column	1	2	3	4
1	0	1	0	1
2	0	0	0	0
3	0	1	1	1
4	0	1	0	0

2 GRAPH REPRESENTATION
ADJACENCY LIST

GRAPH REPRESENTATION

ADJACENCY LIST

WEIGHTED GRAPH

FOR EACH VERTEX STORE A LIST OF VERTICES ADJACENT TO V AND THE WEIGHT

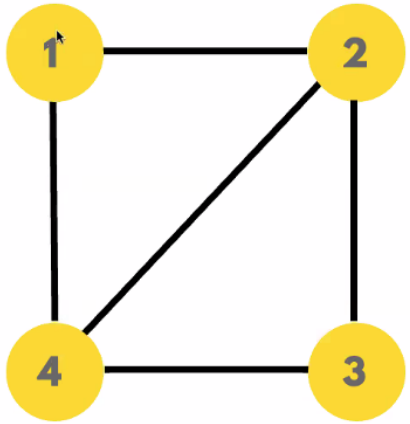
UNWEIGHTED GRAPH

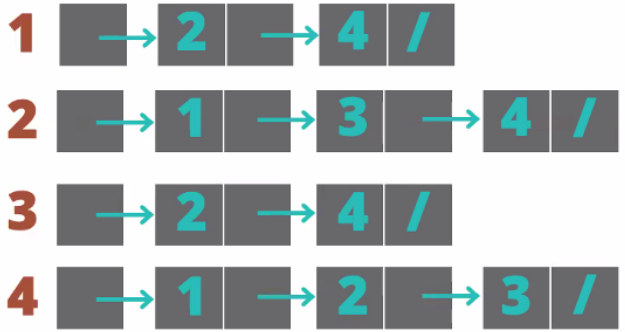
FOR EACH VERTEX STORE A LIST OF VERTICES ADJACENT TO V

GRAPH REPRESENTATION

ADJACENCY LIST

UNDIRECTED GRAPH EXAMPLE

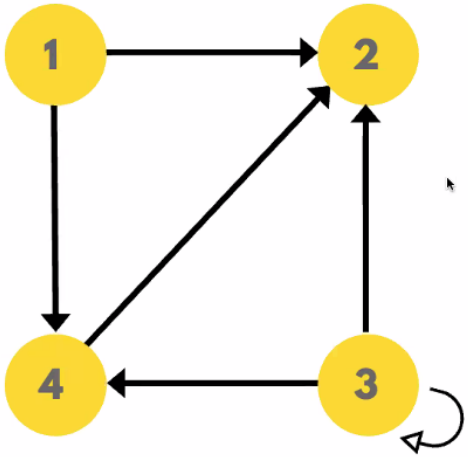


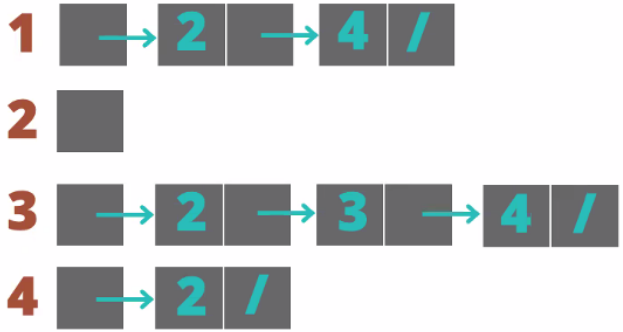


GRAPH REPRESENTATION

ADJACENCY LIST

DIRECTED GRAPH EXAMPLE

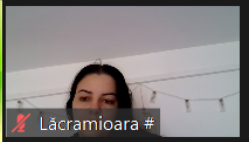


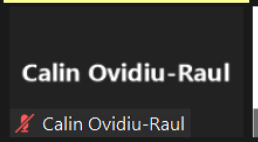


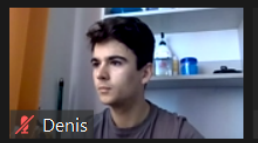
GRAPH REPRESENTATION

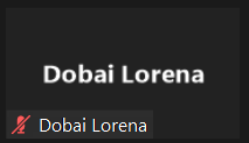
1.ADJACENCY MATRIX

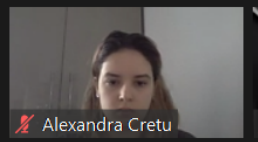
2.ADJACENCY LIST

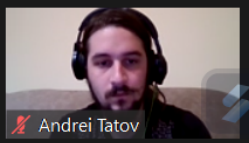


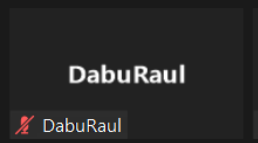


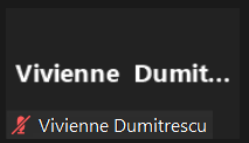








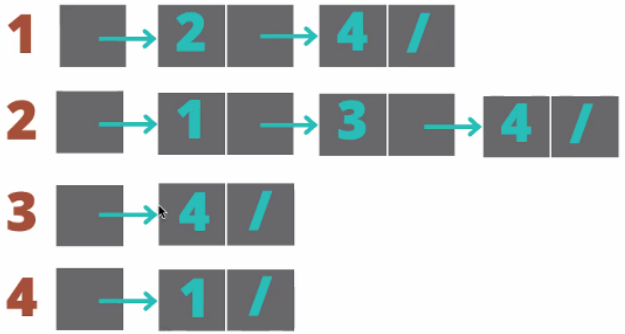




GRAPH REPRESENTATION

EXERCISE

Transform the adjacency list to a adjacency matrix



row \ column	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	0	0	1
4	1	0	0	0

SPANNING TREE

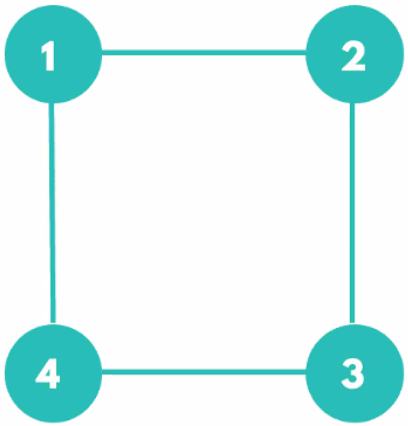
DEFINITION

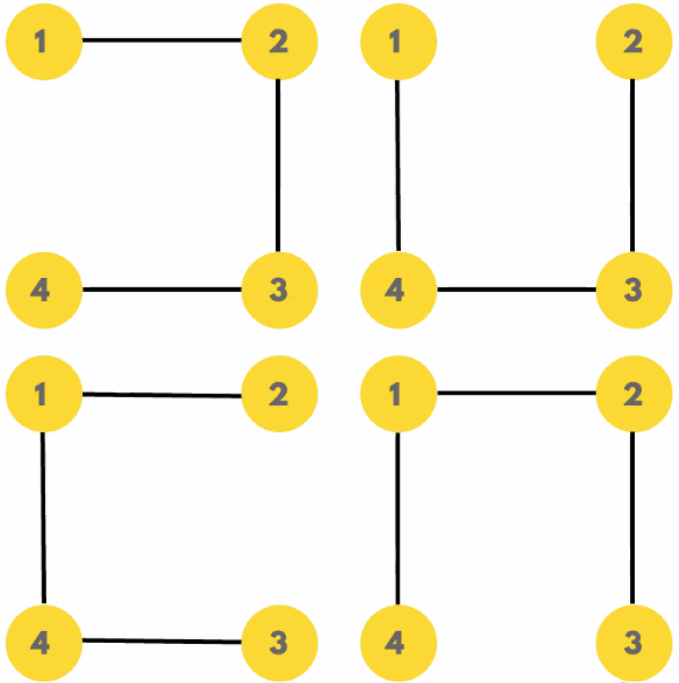
A **spanning tree** is a **sub-graph** of an **undirected connected** graph, which includes all the **vertices** of the graph with a minimum possible number of **edges**.

THE EDGES MAY OR MAY NOT HAVE WEIGHTS ASSIGNED TO THEM.

SPANNING TREE

EXAMPLE



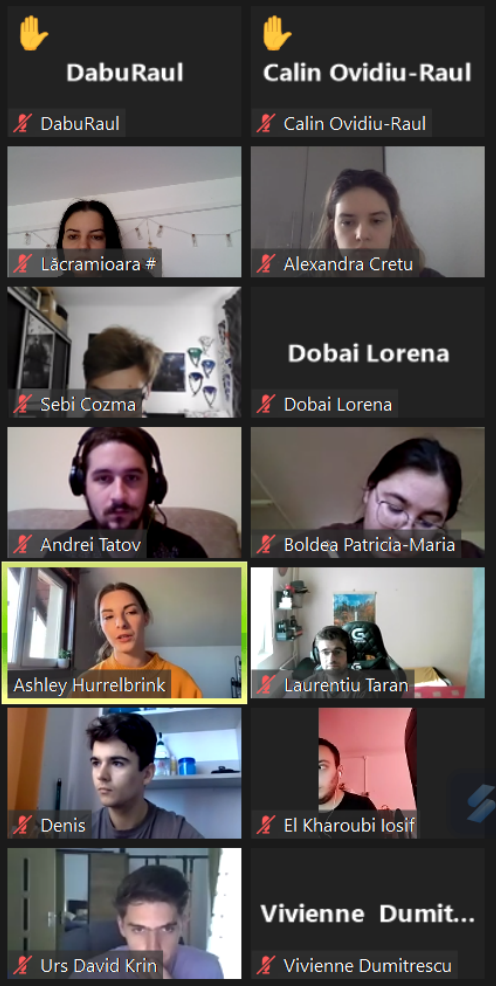


MINIMUM SPANNING TREE

DEFINITION

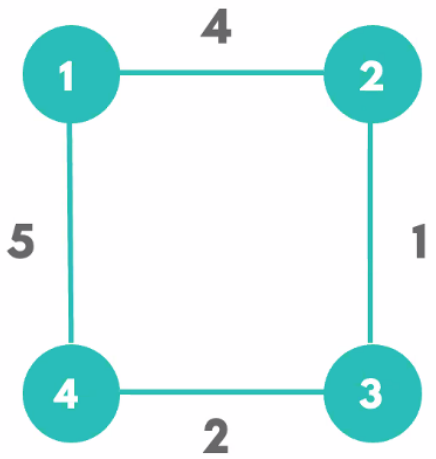
A **minimum spanning tree** is a spanning tree in which the **sum of the weight of the edges (COST)** is as minimum as possible.

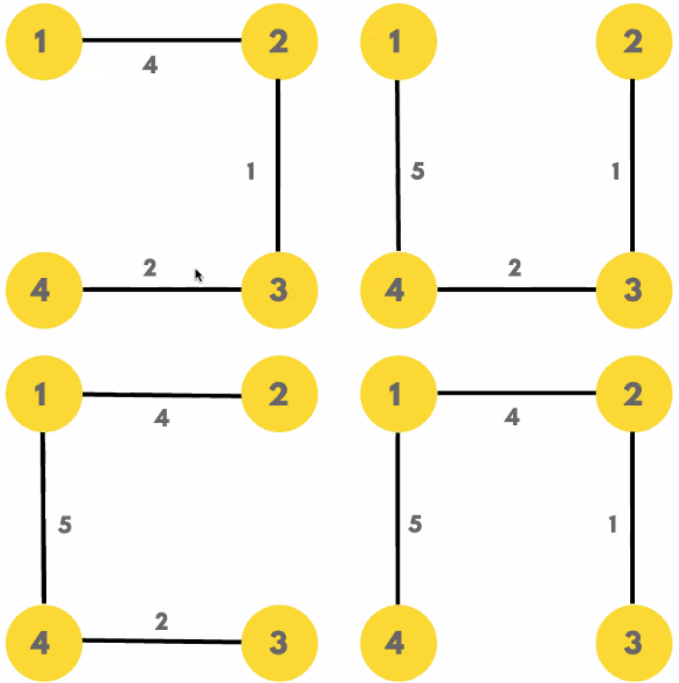
THE EDGES MUST HAVE WEIGHTS ASSIGNED TO THEM.



MINIMUM SPANNING TREE

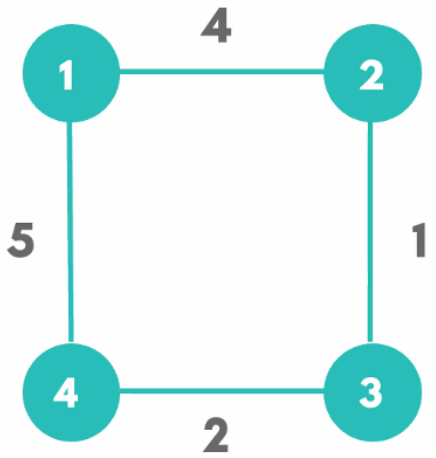
EXAMPLE

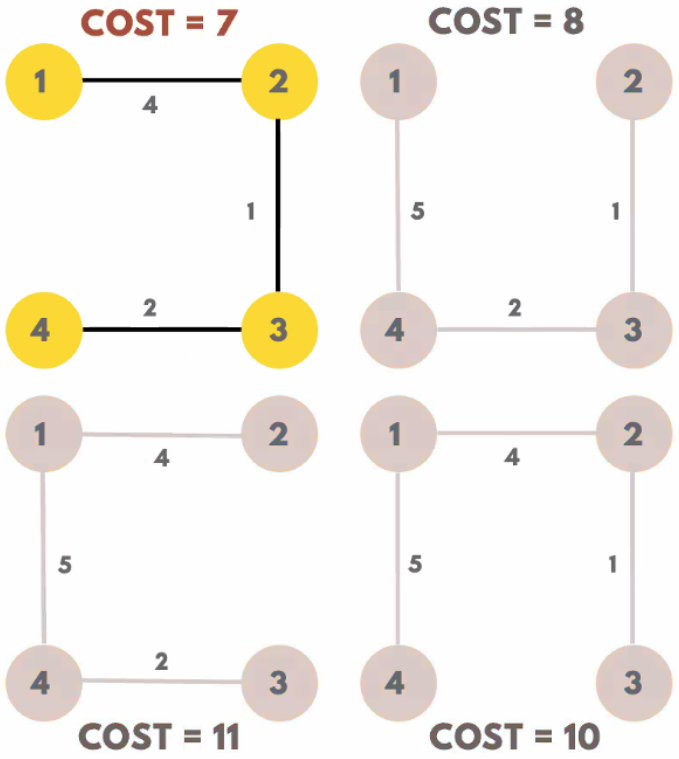




MINIMUM SPANNING TREE

EXAMPLE



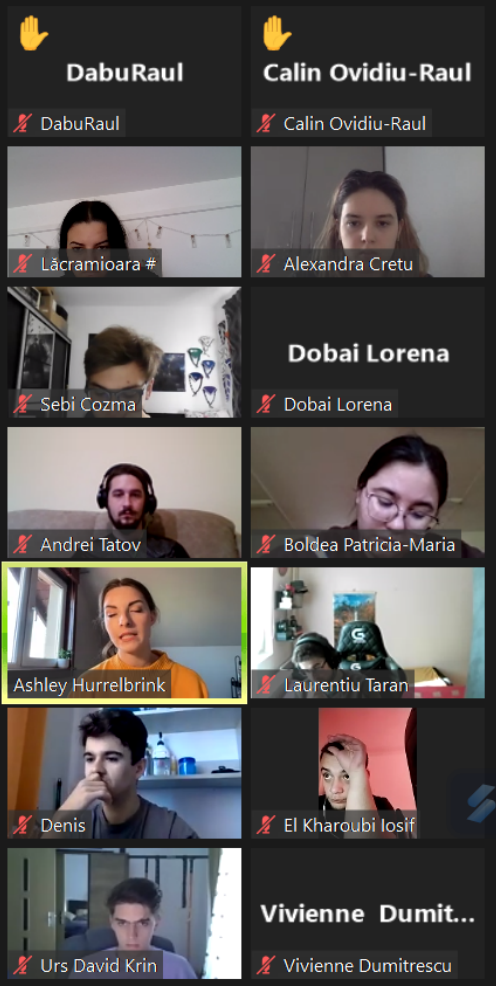


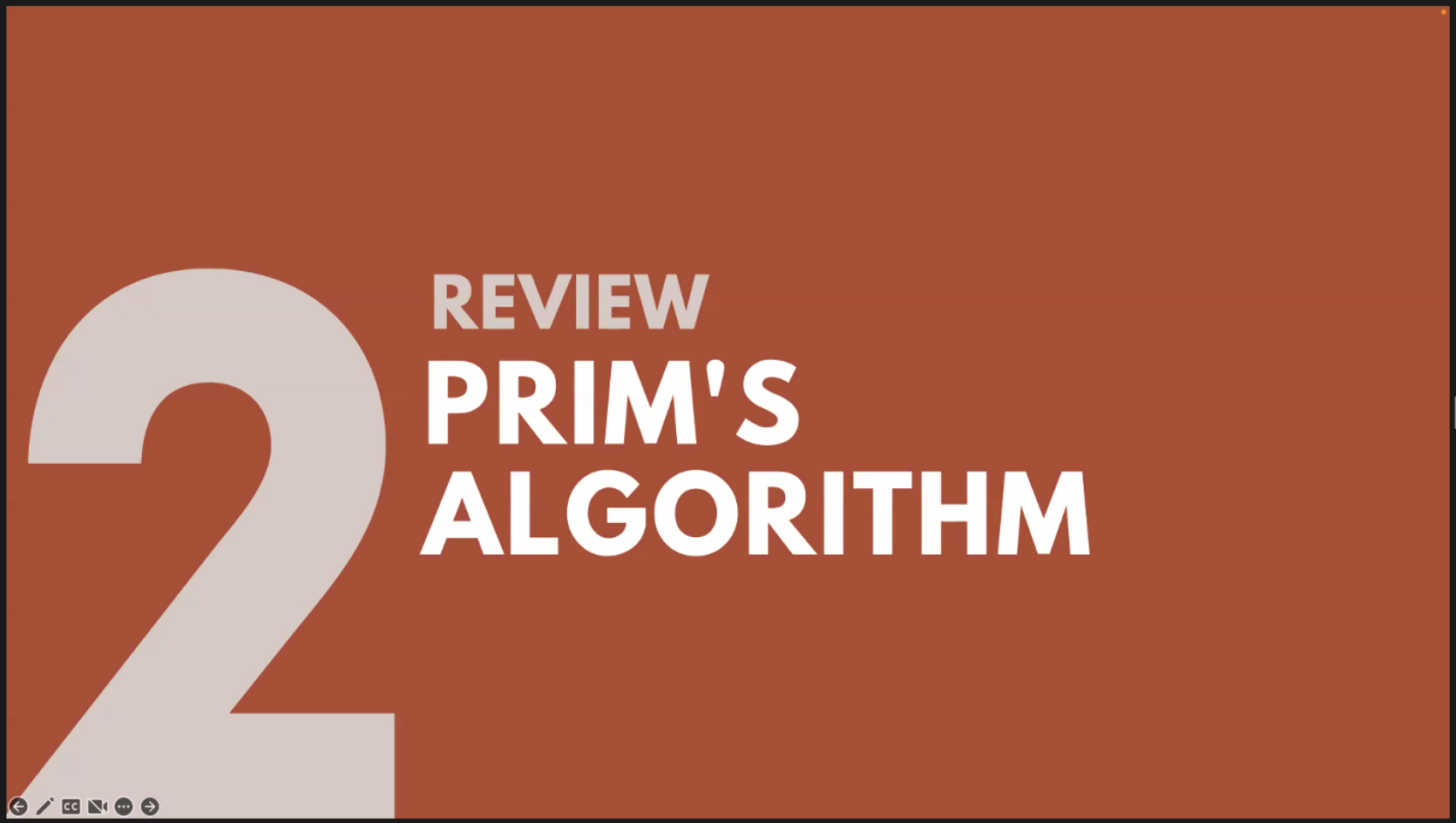
MINIMUM SPANNING TREE

PROPERTIES

THE MST OF A GRAPH WITH N NODES MUST:

- **contain all the n nodes**
- **contain N-1 edges**
- **provide exactly one path between any two nodes of the graph**





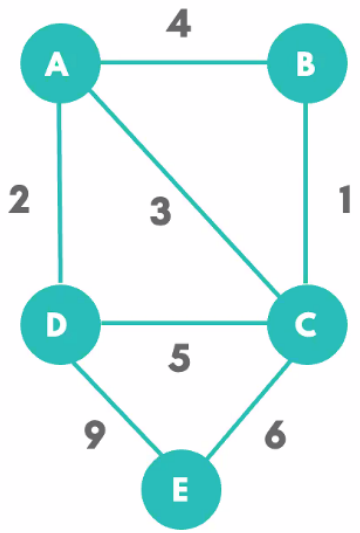
PRIM'S ALGORITHM

STEPS

1. INITIALIZE THE MINIMUM SPANNING TREE WITH A VERTEX CHOSEN AT RANDOM.
2. FIND ALL THE EDGES THAT CONNECT THE TREE TO NEW VERTICES, FIND THE MINIMUM AND ADD IT TO THE TREE.
3. KEEP REPEATING STEP 2 UNTIL WE GET A MINIMUM SPANNING TREE.

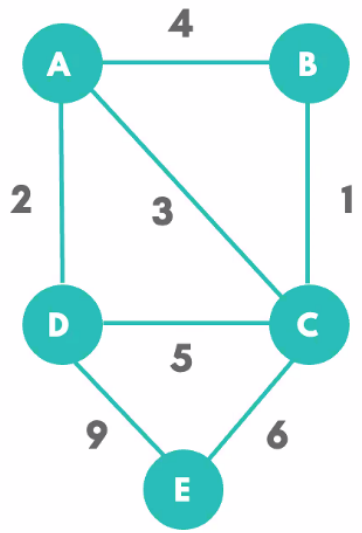
PRIM'S ALGORITHM

STEPS



PRIM'S ALGORITHM

STEPS



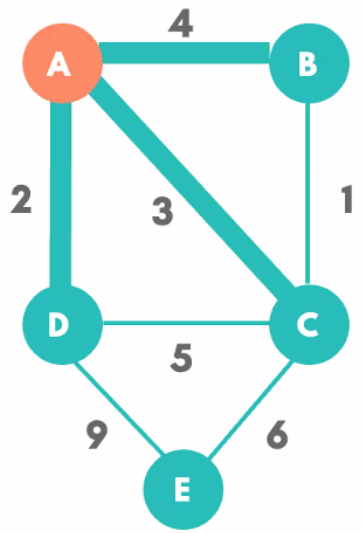
MST

ORDER

1 INITIALIZE THE MINIMUM SPANNING TREE WITH A VERTEX CHOSEN AT RANDOM.

PRIM'S ALGORITHM

STEPS



MST

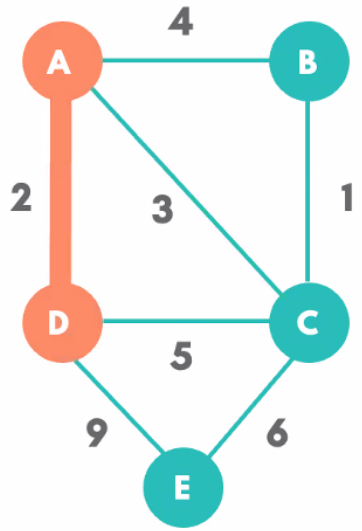
{ A }

ORDER

2 FIND ALL THE EDGES THAT CONNECT THE TREE TO NEW VERTICES, FIND THE MINIMUM AND ADD IT TO THE TREE.

PRIM'S ALGORITHM

STEPS



MST

{ A, D }

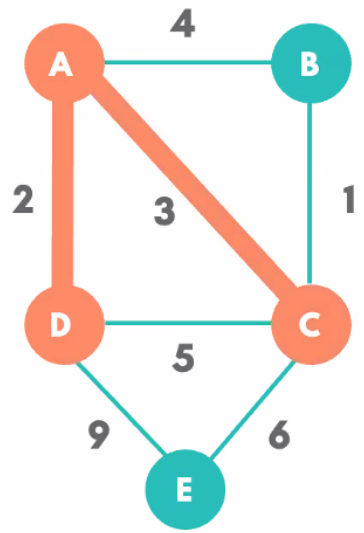
ORDER

(A - D)

2 FIND ALL THE EDGES THAT CONNECT THE TREE TO NEW VERTICES, FIND THE MINIMUM AND ADD IT TO THE TREE.

PRIM'S ALGORITHM

STEPS



MST

{ A, D, C}

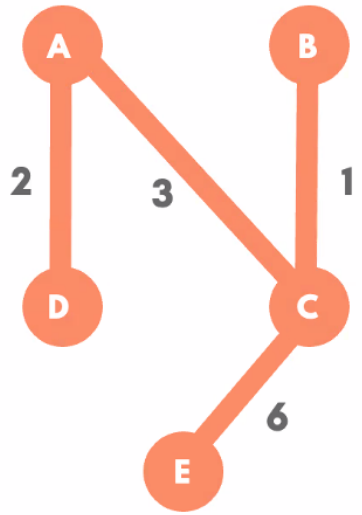
ORDER

(A - D)
(A - C)

2 FIND ALL THE EDGES THAT CONNECT THE TREE TO NEW VERTICES, FIND THE MINIMUM AND ADD IT TO THE TREE.

PRIM'S ALGORITHM

STEPS



MST

{ A, D, C, B, E}

ORDER

(A - D)
(A - C)
(C - B)
(C - E)

3 KEEP REPEATING STEP 2 UNTIL WE GET A MINIMUM SPANNING TREE.

PRIM'S ALGORITHM

IMPLEMENTATION

Given G = (V, E). Output a MST.

O(V*X)

Randomly select a vertex v

VISIT= {v};
MST = {};

While (VISIT != V)

Find a vertex u∈V-VISIT that connects to a vertex v∈VISIT such that w(u,v) ≤ w(x,y), for any x∈V-VISIT and y∈VISIT

O(X)

VISIT = VISIT U {u};
MST = MST U (u, v)

EndWhile

Return A

PRIM'S ALGORITHM

BRUTE FORCE

Test all edges

Find a vertex u∈V-VISIT that connects to a vertex v∈VISIT such that w(u,v) ≤ w(x,y), for any x∈V-VISIT and y∈VISIT

O(X)

```
min_weight = infinity
For each edge (x, y) in E
    if x∈ V-VISIT   &&  y∈ VISIT  && w(x, y) < min_weight
        u = x;
        v = y;
        min_weight = w(x, y);
```

O(E)

PRIM'S ALGORITHM

BRUTE FORCE

Test all edges

Find a vertex u∈V-VISIT that connects to a vertex v∈VISIT such that w(u,v) ≤ w(x,y), for any x∈V-VISIT and y∈VISIT

O(X)

```
min_weight = infinity
For each edge (x, y) in E
    if x∈ V-VISIT   &&  y∈ VISIT  && w(x, y) < min_weight
        u = x;
        v = y;
        min_weight = w(x, y);
```

O(V^2)

PRIM'S ALGORITHM

BRUTE FORCE IMPLEMENTATION

Given G = (V, E). Output a MST.

O(V^3)

Randomly select a vertex v

VISIT= {v};
MST = {};

While (VISIT != V)

Find a vertex u∈V-VISIT that connects to a vertex v∈VISIT such that w(u,v) ≤ w(x,y), for any x∈V-VISIT and y∈VISIT

O(V^2)

VISIT = VISIT U {u};
MST = MST U (u, v)

EndWhile

Return A

PRIM'S ALGORITHM

IMPLEMENTATION

Given G = (V, E). Output a MST.

Find a vertex u∈V-VISIT that connects to a vertex v∈VISIT such that w(u,v) ≤ w(x,y), for any x∈V-VISIT and y∈VISIT

O(X)

BRUTE FORCE

Test all edges

IMPROVED

Keep a list of vertex candidates in an array

BETTER

Keep vertex candidates in a priority queue

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```

O(?)

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array                                   O(V)
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array                                O(V)
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).                                             O(V^2)
    for loop to search d and select minimum u with d[u]>0  ->  O(V)

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0

    //update distance and parent array
    for all edges from u to w                                   O(V)
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```

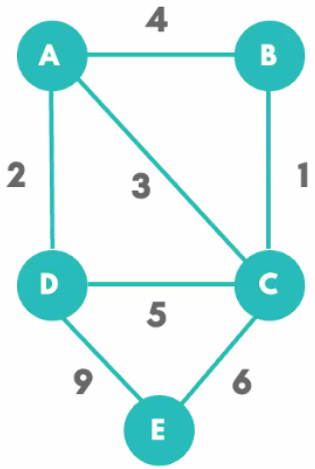
O(?)

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



VISIT

MST

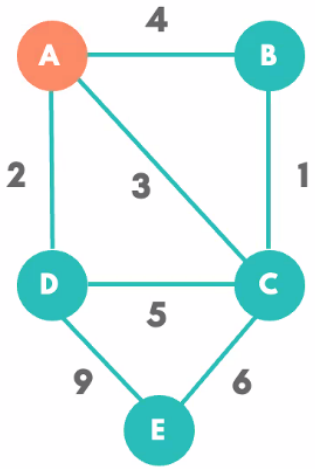
	A	B	C	D	E
d	inf	inf	inf	inf	inf
p	null	null	null	null	null

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



VISIT

MST

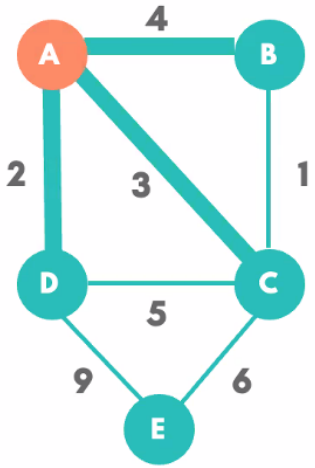
	A	B	C	D	E
d	0	inf	inf	inf	inf
p	null	null	null	null	null

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



VISIT

MST

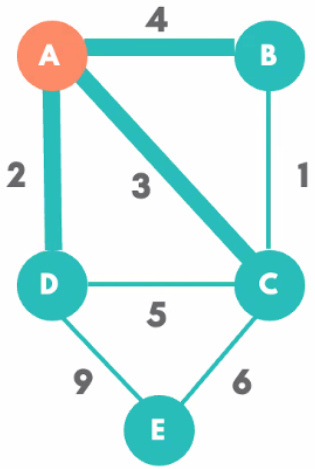
	A	B	C	D	E
d	0	4	3	2	inf
p	null	A	A	A	null

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



VISIT {A}

MST { }

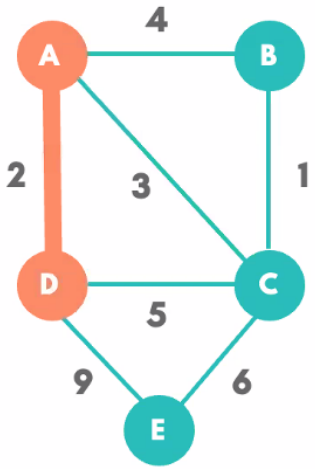
	A	B	C	D	E
d	0	4	3	2	inf
p	null	A	A	A	null

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u D

VISIT {A, D}

MST {A-D}

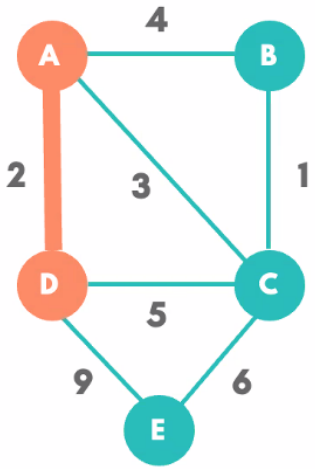
	A	B	C	D	E
d	0	4	3	2	inf
p	null	A	A	A	null

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u D

VISIT {A, D}

MST {A-D}

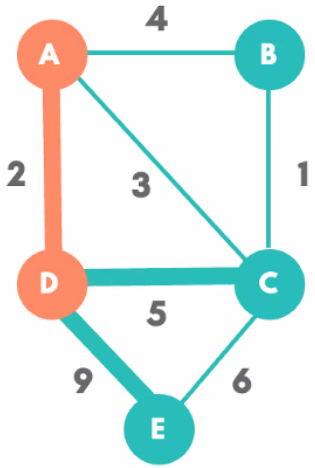
	A	B	C	D	E
d	0	4	3	0	inf
p	null	A	A	A	null

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u D

VISIT {A, D}

MST {A-D}

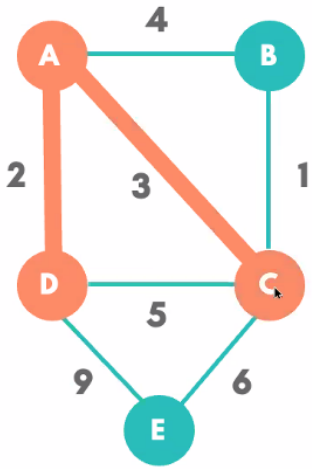
	A	B	C	D	E
d	0	4	3	0	9
p	null	A	A	A	E

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u C

VISIT {A , D , C}

MST {A-D , A-C }

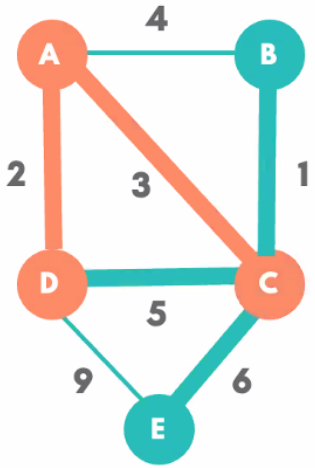
	A	B	C	D	E
d	0	4	3	0	9
p	null	A	A	A	E

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u C

VISIT {A , D , C}

MST {A-D , A-C }

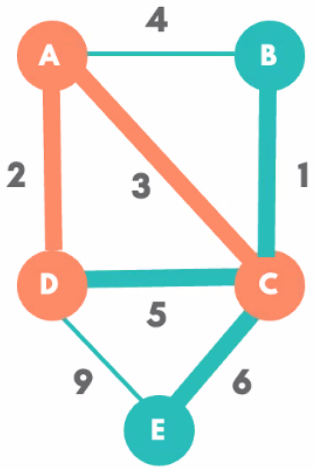
	A	B	C	D	E
d	0	1	0	0	6
p	null	C	A	A	C

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u C

VISIT {A , D , C}

MST {A-D , A-C }

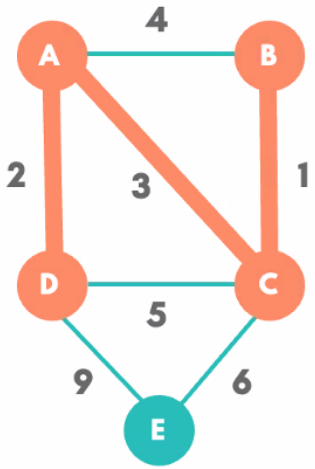
	A	B	C	D	E
d	0	1	0	0	6
p	null	C	A	A	C

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u B

VISIT {A , D , C , B}

MST {A-D , A-C , C-B }

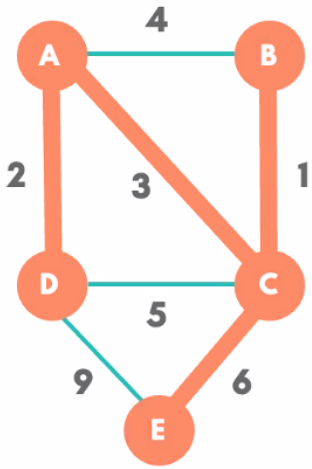
	A	B	C	D	E
d	0	0	0	0	6
p	null	C	A	A	C

PRIM'S ALGORITHM

DISTANCE ARRAY

Keep a list of vertex candidates in an array

```
//set distance and parent array
For each v in V
    d[v] = infinity
    p[v] = null

d[v] = 0 //v is randomly chosen

//update distance and parent array
For all edges from v to u
    d[u] = weight(edge(v,u))
    p[u] = v

VISIT = {v}
MST = {}

While (VISIT != V).
    for loop to search d and select minimum u with d[u]>0

    VISIT = VISIT U {u}
    MST = MST U {edge(p[u], u)

    d[u] = 0;

    //update distance and parent array
    for all edges from u to w
        if(weight(edge(u,w)) < d[u])
            d[w] = weight(edge(u, w)
            p[w] = u;
```



u E

VISIT {A, D, C, B, E}

MST {A-D, A-C, C-B, C-E}

	A	B	C	D	E
d	0	0	0	0	0
p	null	C	A	A	C

PRIM'S ALGORITHM

IMPLEMENTATION

Given G = (V, E). Output a MST.

Find a vertex $u \in$ V-VISIT that connects to a vertex $v \in$ VISIT such that $w(u,v) \leq w(x,y)$, for any $x \in$ V-VISIT and $y \in$ VISIT

O(X)

BRUTE FORCE	IMPROVED	BETTER
Test all edges	Keep a list of vertex candidates in an array	Keep vertex candidates in a priority queue

PRIM'S ALGORITHM

PRIORITY QUEUE

Keep a list of vertex candidates in an array

```
//set priority queue, key and parent array
For each u in Q                                          → O(V)
    Q.insert(u)
    key[u] = infinity
    p[u] = null

DECREASE_KEY(v, 0) //v is randomly chosen

VISIT = { v }
MST = { }

While (Q not Empty).                                     → O(V)
    u = Q.extractMin();                                  → O(log V)

    //update key and parent array
    for all edges from u to w                            → O(V)
        if(Q.contains(u) && weight(edge(u,w)) < key[u])
            DECREASE_KEY(u, weight(edge(u, w)))          → O(log V)
            p[w] = u;
```

O(V^2*log V)

PRIM'S ALGORITHM

IMPLEMENTATION

Given G = (V, E). Output a MST.

DISTANCE ARRAY

$O(V*V)$

Good for dense graphs

PRIORITY QUEUE

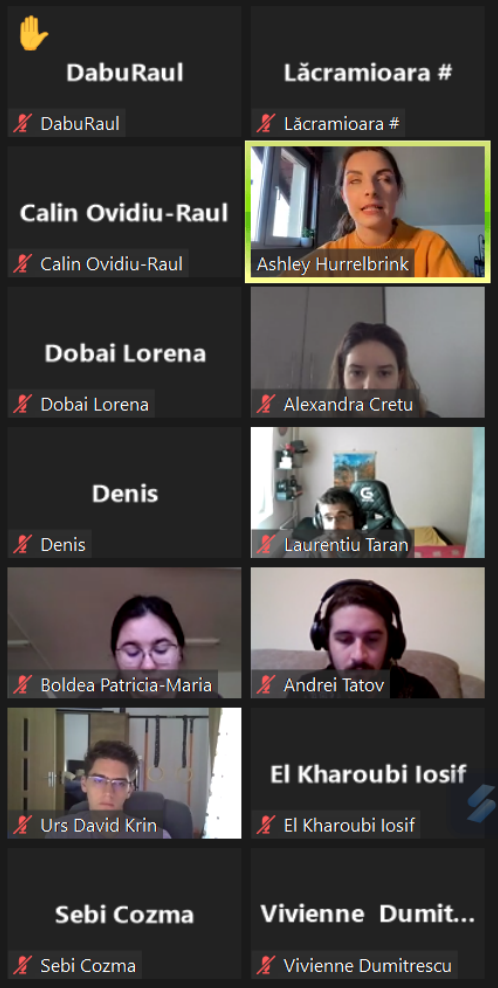
$O(E * \log V)$

Good for sparse graphs

KRUSKAL'S ALGORITHM

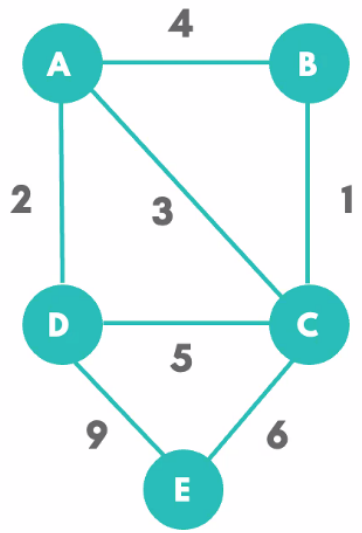
STEPS

1. Sort all the edges in non-decreasing order of their weight.
2. PICK THE SMALLEST EDGE. CHECK IF IT FORMS A CYCLE WITH THE SPANNING TREE FORMED SO FAR. IF CYCLE IS NOT FORMED, INCLUDE THIS EDGE. ELSE, DISCARD IT.
3. REPEAT STEP#2 UNTIL THERE ARE (V-1) EDGES IN THE SPANNING TREE.



KRUSKAL'S ALGORITHM

STEPS



(B-C) --> 1
(A-D) --> 2
(A-C) --> 3
(A-B) --> 4
(D-C) --> 5
(C-E) --> 6
(D-E) --> 9

MST

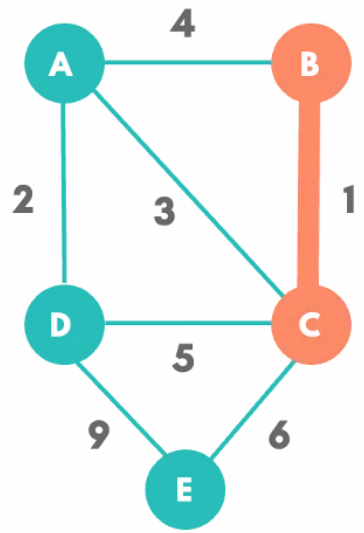
{A},{B},{C},{D},{E}

ORDER

1 Sort all the edges in non-decreasing order of their weight.

KRUSKAL'S ALGORITHM

STEPS



(B-C) --> 1
(A-D) --> 2
(A-C) --> 3
(A-B) --> 4
(D-C) --> 5
(C-E) --> 6
(D-E) --> 9

MST

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

ORDER

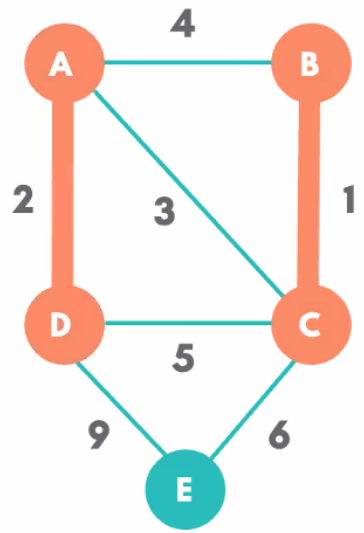
(B - C)

2 PICK THE SMALLEST EDGE. CHECK IF IT FORMS A CYCLE WITH THE SPANNING TREE FORMED SO FAR. IF CYCLE IS NOT FORMED, INCLUDE THIS EDGE. ELSE, DISCARD IT.

KRUSKAL'S ALGORITHM

STEPS

(B-C) --> 1
(A-D) --> 2
(A-C) --> 3
(A-B) --> 4
(D-C) --> 5
(C-E) --> 6
(D-E) --> 9



MST

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

ORDER

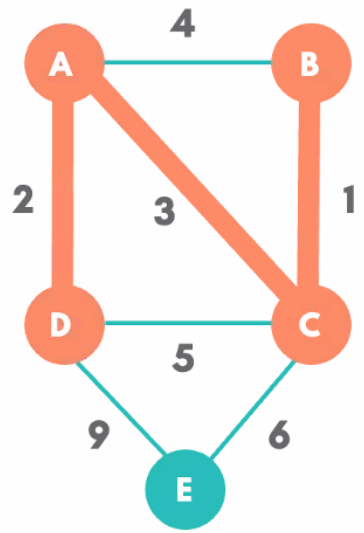
(B - C)
(A - D)

2 PICK THE SMALLEST EDGE. CHECK IF IT FORMS A CYCLE WITH THE SPANNING TREE FORMED SO FAR. IF CYCLE IS NOT FORMED, INCLUDE THIS EDGE. ELSE, DISCARD IT.

KRUSKAL'S ALGORITHM

STEPS

(B-C) --> 1
(A-D) --> 2
(A-C) --> 3
(A-B) --> 4
(D-C) --> 5
(C-E) --> 6
(D-E) --> 9



MST

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

ORDER

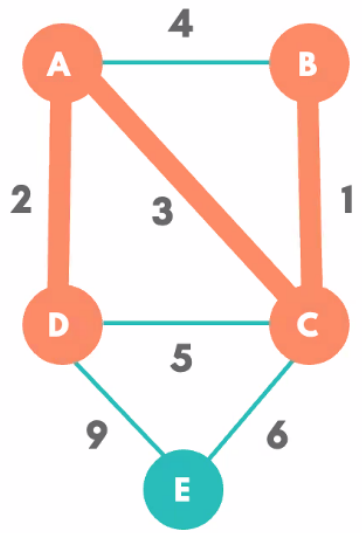
(B - C)
(A - D)
(A - C)

2 PICK THE SMALLEST EDGE. CHECK IF IT FORMS A CYCLE WITH THE SPANNING TREE FORMED SO FAR. IF CYCLE IS NOT FORMED, INCLUDE THIS EDGE. ELSE, DISCARD IT.

KRUSKAL'S ALGORITHM

STEPS

(B-C) --> 1
(A-D) --> 2
(A-C) --> 3
~~(A-B) --> 4~~
(D-C) --> 5
(C-E) --> 6
(D-E) --> 9



MST

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

ORDER

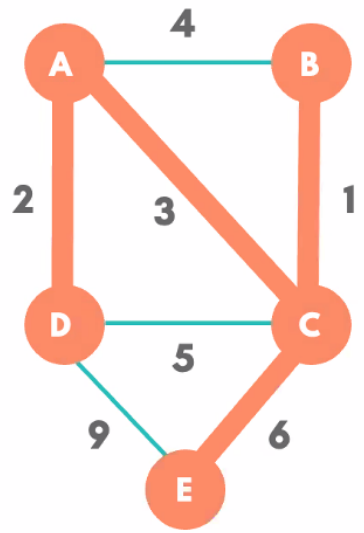
(B - C)
(A - D)
(A - C)

2 PICK THE SMALLEST EDGE. CHECK IF IT FORMS A CYCLE WITH THE SPANNING TREE FORMED SO FAR. IF CYCLE IS NOT FORMED, INCLUDE THIS EDGE. ELSE, DISCARD IT.

KRUSKAL'S ALGORITHM

STEPS

(B-C) --> 1
(A-D) --> 2
(A-C) --> 3
~~(A-B) --> 4~~
~~(D-C) --> 5~~
(C-E) --> 6
(D-E) --> 9



MST

{ A , D , B , C , E }

ORDER

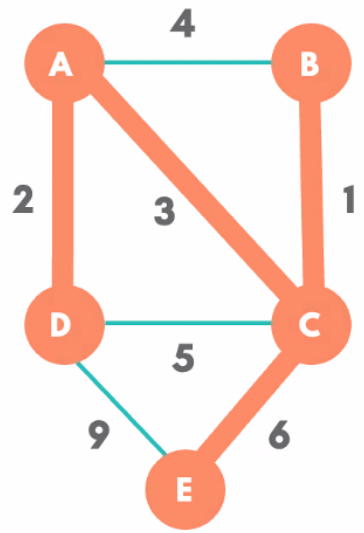
(B - C)
(A - D)
(A - C)
(C - E)

2 PICK THE SMALLEST EDGE. CHECK IF IT FORMS A CYCLE WITH THE SPANNING TREE FORMED SO FAR. IF CYCLE IS NOT FORMED, INCLUDE THIS EDGE. ELSE, DISCARD IT.

KRUSKAL'S ALGORITHM

STEPS

(B-C) --> 1
(A-D) --> 2
(A-C) --> 3
~~(A-B) --> 4~~
~~(D-C) --> 5~~
(C-E) --> 6
(D-E) --> 9



MST

{ A , D , B , C , E }

ORDER

(B - C)
(A - D)
(A - C)
(C - E)

3 REPEAT STEP#2 UNTIL THERE ARE (V-1) EDGES IN THE SPANNING TREE.

2 KRUSKAL'S ALGORITHM IMPLEMENTATION

KRUSKAL'S ALGORITHM

IMPLEMENTATION

Given G = (V, E). Output a MST.

```
MST = {}

//create tree and add it to the forest MST
For each v in V
    MAKE_SET(v)

SORT(e)  //sort all the edges into increasing order by weight

//find edge with lowest cost and unite the trees
For all edges(u, v) in increasing order
    if ( FIND_SET(u) != FIND_SET(v) )
        UNION(u, v)
        MST = MST U {(u,v)}
```

KRUSKAL'S ALGORITHM

IMPLEMENTATION

Given G = (V, E). Output a MST.

MAKE_SET()

FIND_SET()

UNION()

LINKED LIST

Each set is a singly linked list

AS A FOREST

Forests of up-trees are used

KRUSKAL'S ALGORITHM

LINKED LIST

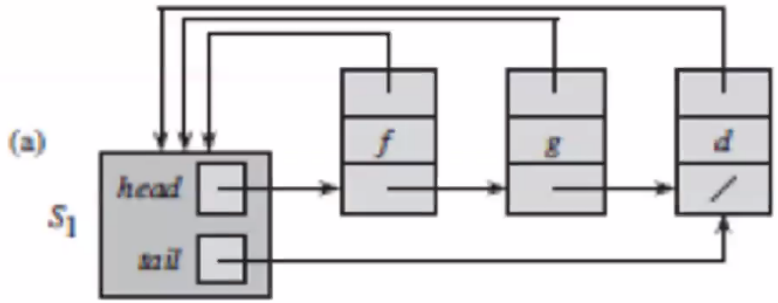
Given G = (V, E). Output a MST.

SET ATTRIBUTES

- Head
- Tail

OBJECT ATTRIBUTES

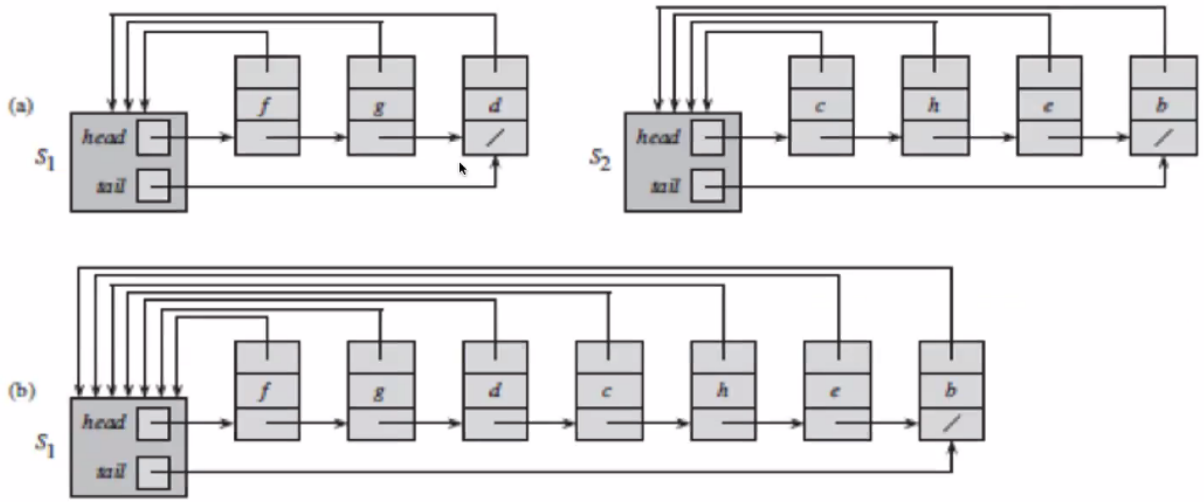
- the set member
- pointer to the set object
- next



KRUSKAL'S ALGORITHM

LINKED LIST

Given G = (V, E). Output a MST.



KRUSKAL'S ALGORITHM

LINKED LIST

Given G = (V, E). Output a MST.

MAKE_SET(X)

Create a new **linked list** whose only object is **x**.

FIND_SET(X)

Follow the pointer from x back to its set object and then return the member in the object that **head points to**.

UNION(X,Y)

Append y's list onto the end of x's list.

The representative of x's list becomes the new **set representative**.

Use the **tail pointer** for x's list to quickly find where to append y's list.

KRUSKAL'S ALGORITHM

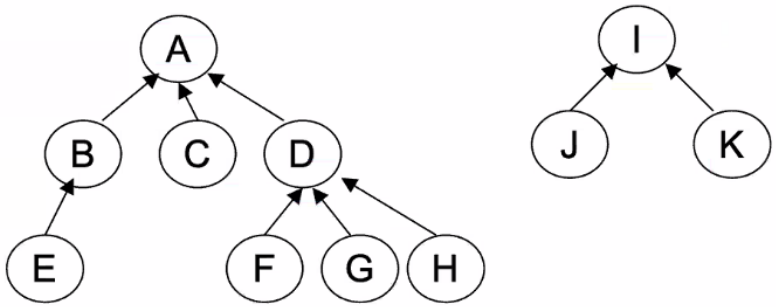
AS A FOREST

Given G = (V, E). Output a MST.

IMPLEMENTED AS A FOREST OF UP-TREES

UP-TREE

- Set of nodes
- Every node has a parent except for the root
- The root is the set representative



KRUSKAL'S ALGORITHM

AS A FOREST

Given G = (V, E). Output a MST.

MAKE_SET(X)

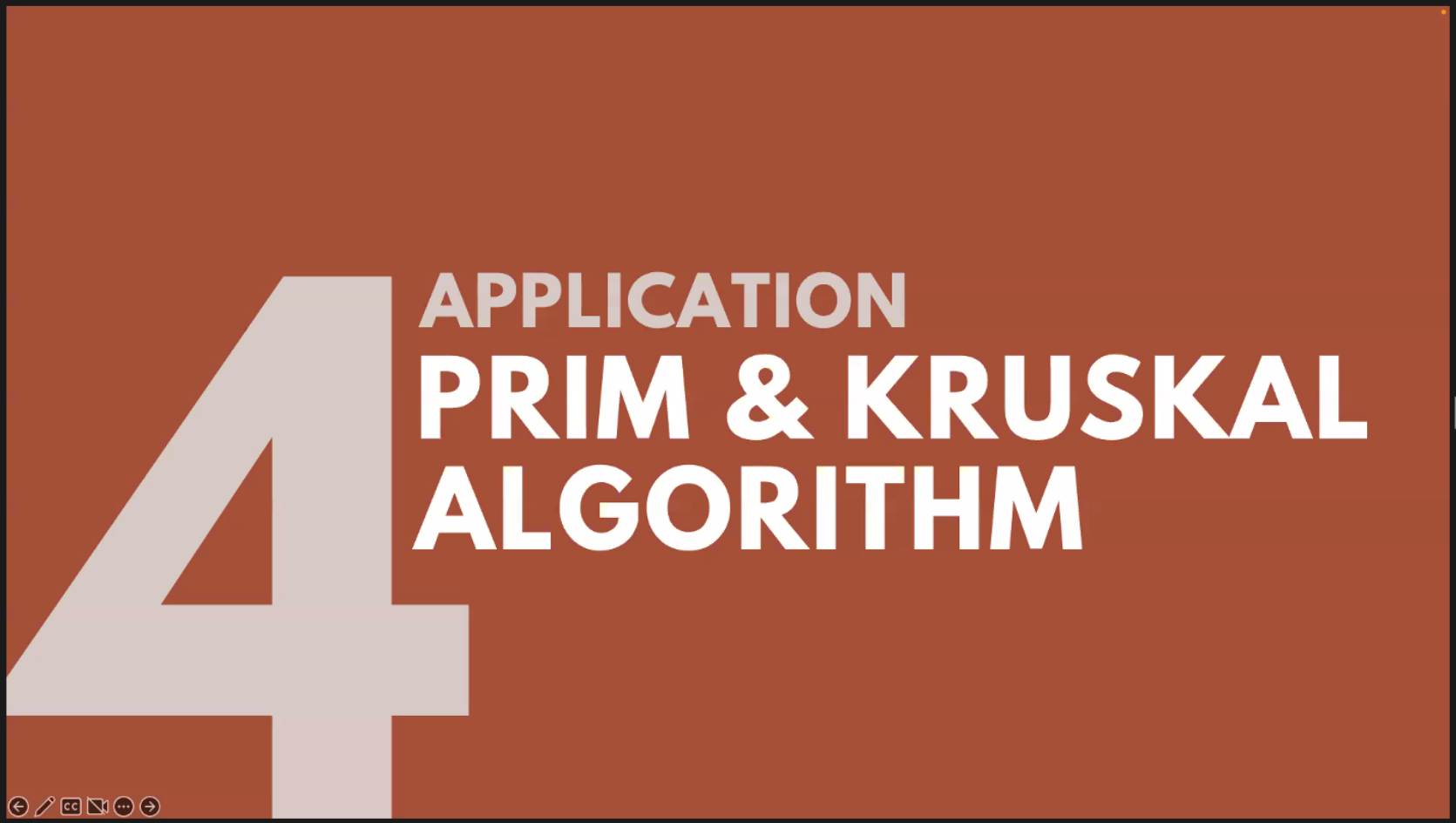
Initialize node as root

FIND_SET(X)

walk upwards in the tree, starting from x, following parent links, until arriving at the root.

UNION(X,Y)

One tree becomes a subtree of the other



OVERVIEW

PRIM VS KRUSKAL

Prim's Algorithm	Kruskal's Algorithm
The tree that we are making or growing always remains connected.	The tree that we are making or growing usually remains disconnected.
Prim's Algorithm grows a solution from a random vertex by adding the next cheapest vertex to the existing tree.	Kruskal's Algorithm grows a solution from the cheapest edge by adding the next cheapest edge to the existing tree / forest.
Prim's Algorithm is faster for dense graphs.	Kruskal's Algorithm is faster for sparse graphs.

LAB 10 Application

PRIM & KRUSKAL

1. IMPLEMENT PRIM'S ALGORITHM FOR MST USING DISTANCE ARRAYS.

2. GIVEN THE WEIGHTED UNDIRECTED GRAPH IN THE FIGURE, ILLUSTRATE THE STEPS OF PRIM'S ALGORITHM. APPLY THE ALGORITHM ILLUSTRATING STEP BY STEP THE CONTENTS OF THE DATA STRUCTURES THAT ARE USED.

3. GIVEN THE WEIGHTED UNDIRECTED GRAPH, ILLUSTRATE THE STEPS OF KRUSKAL'S ALGORITHM. EXPLAIN EACH STEP OF THE ALGORITHM AND THE CONTENTS OF THE ADDITIONAL DATA STRUCTURES THAT ARE USED.