Lecture - 3

MINIMUM SPANNING TREE

Spanning Tree

Given a connected *undirected* graph $G = (V, E)$, $T = (V_T, E_T)$ is a spanning tree if $V_T = V$, $E_T \subseteq E$, T is acyclic, and T is connected.

G can only have a spanning tree if it is connected.

The number of edges in a spanning tree is $|V| - 1$.

T is a tree because it is acyclic.

T is spanning because it every vertex from G.

Minimum Spanning Tree

If weights are associated with each edge on graph G, then each spanning tree $T = (V', E')$ has a weight which is the total weight of each edge in E'.

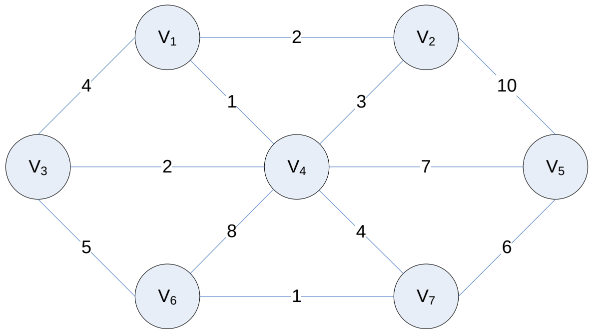
The minimum spanning tree is the spanning tree with weight less than or equal to every other spanning tree.

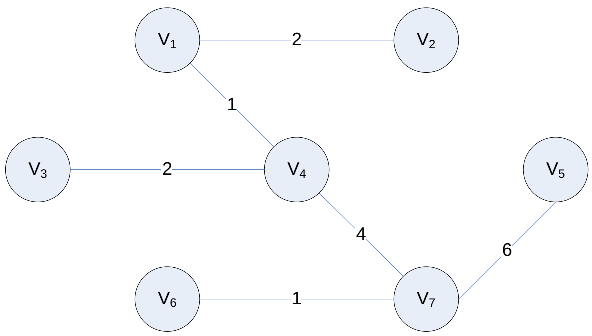
A graph may have multiple minimum spanning trees.

If all the edges in a graph have the same weight, the all spanning trees are minimum.

If each edge has a distinct weight, then there will be only one minimum spanning tree.

Example (a graph and its minimum spanning tree)





❖ ***Prim's Algorithm***

Prim's algorithm is similar to Dijkstra's algorithm for shortest paths, except distance is the weight of the shortest edge connection an unknown vertex to a known vertex (in Dijkstra's, it was a sum).

After a vertex v is selected, for each unknown w adjacent to v, $w.\text{distance} = \min(w.\text{distance}, c_{v,w})$.

The general strategy

- Choose a starting vertex, say v_1 and mark it as known.

  The path length from v_1 to v_1 is 0.

  Find all vertices adjacent to v_1. These are v_2, v_3, and v_4.

  Adjust the distance and vertex for v_2, v_3, and v_4 (e.g. $v_2.\text{distance} = c_{1,2} = 2$ and $v_2.\text{pathVertex} = v_1$).

- Find the unknown vertex with the smallest distance. Select v_4 and mark it as known.

  Find all the vertices adjacent to v_4. These are v_1, v_2 v_3, v_5, v_6, and v_7.

  v_1 is already known, so no change.

Adjust the distance and vertex for v_2 v_3, v_5, v_6, *and* v_7 *(e.g.* $v_3.\text{distance} = c_{4,3} = 2$ *and* $v_3.\text{pathVertex} = v_4$*)*

For v_2, $v_2.\text{distance} = c_{4,2} = 3$. *Since* $v_2.\text{distance} = 2$, *then no change to* v_2

- Find the unknown vertex with the smallest distance (could be either v_2 or v_3). Select v_2 and mark it as known.

Find all vertices adjacent to v_2. *These are* v_1, v_4 *and* v_5.

v_1 *and* v_4 *are already known, so no change.*

Adjust the distance and vertex for v_5. $v_5.\text{distance} = c_{2,5} = 10$. *Since* $v_5.\text{distance} = 7$, *then no change to* v_5.

- Find the unknown vertex with the smallest distance. Select v_3 and mark it as known.

Find all the vertices adjacent to v_3. *These are* v_1, v_4 *and* v_6.

v_1 *and* v_4 *are already known, so no change.*

Adjust the distance and vertex for v_6. $v_6.\text{distance} = c_{3,6} = 5$. *Since* $v_6.\text{distance} = 8$, *then* $v_6.\text{distance} = 5$ *and* $v_6.\text{pathVertex} = v_3$.

- Find the unknown vertex with the smallest distance. Select v_7 and mark it as known.

Find all the vertices adjacent to v_7. *These are* v_4, v_5 *and* v_6.

v_4 *is known, so no change.*

Adjust the distance and vertex for v_5 *and* v_6.

$v_5.\text{distance} = c_{7,5} = 6$, $v_5.\text{pathVertex} = v_7$

$v_6.\text{distance} = c_{7,6} = 1$, $v_6.\text{pathVertex} = v$

- Find the unknown vertex with the smallest distance. Select v_6 and mark it as known.

Find all the vertices adjacent to v_6. *These are* v_3, v_4 *and* v_7.

These are all known, so no change.

- Find the unknown vertex with the smallest distance. Select v_5 and mark it as known.

Find all the vertices adjacent to v_5. *These are* v_2, v_4 *and* v_7.

These are all known, so no change.

v	known	distance	path vertex
v_1	✓	0	—
v_2	×		
v_3	×		
v_4	×		
v_5	×		
v_6	×		
v_7	×		

v	known	distance	path vertex
v_1	✓	0	—
v_2	✓	2	v_1
v_3	✓	2	v_4
v_4	✓	1	v_1
v_5	✓	6	v_7
v_6	✓	1	v_7
v_7	✓	4	v_4

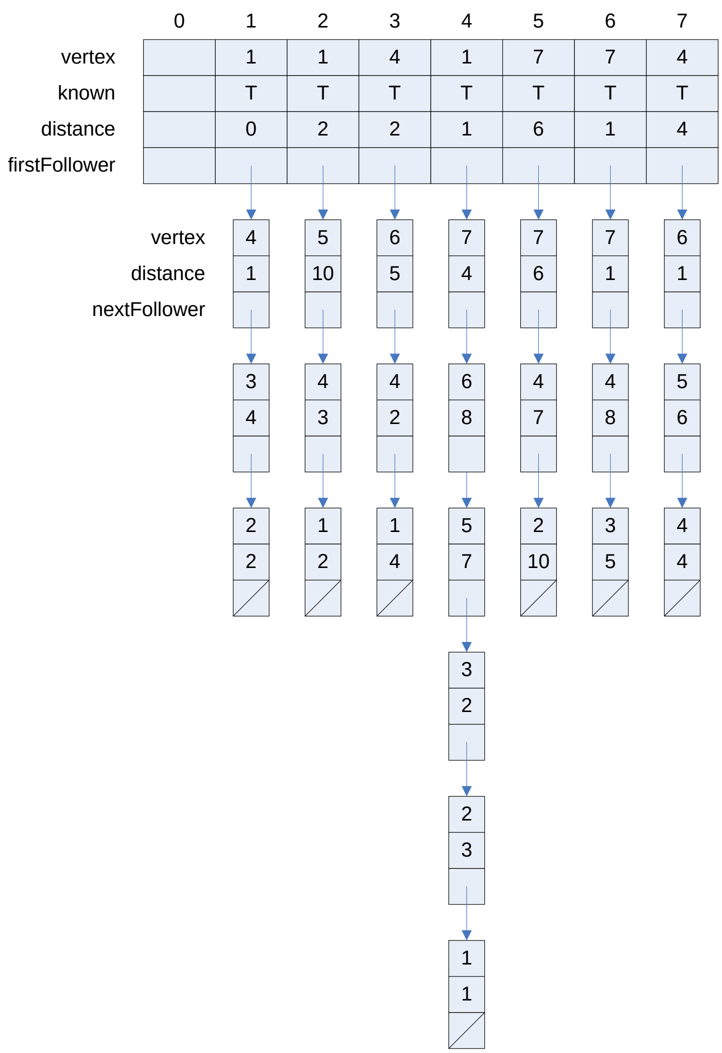
Data Structures

```
const int NO_OF_ELEMENTS = 100;

struct Follower;
struct Follower
{
    Object   vertex;
    int      distance;
    Follower *nextFollower;
};

struct Leader
{
    Object   vertex;
    bool     known;
    int      distance;
    Follower *firstFollower;
};

Leader a[NO_OF_ELEMENTS];
```



Prim’s Minimum Spanning Tree Algorithm

```
PrimsMinimumSpanningTree (startVertex)
for i = 1 to Length(a) – 1
     a[i].vertex = 0
     a[i].known = FALSE
     a[i].distance = HIGH_VALUE
     a[i].firstFollower = NULL
```

```
x = Read(input)
while x != END_OF_INPUT
    y = Read(input)
    p = new Follower Node
    p->vertex = y
    z = Read(input)
    p->distance = z
    p->nextFollower = a[x].firstFollower
    a[x].firstFollower = p
    p = new Follower node
    p->vertex = x
    p->distance = z
    p->nextFollower = a[y].firstFollower
    a[y].firstFollower = p
    x = Read(input)

a[startVertex].vertex = startVertex
a[startVertex].distance = 0

i = startVertex
while 1
    a[i].known = TRUE
    p = a[i].firstFollower
    while p != NULL
        if a[p->vertex].known == FALSE
            if p->distance < a[p->vertex].distance
                a[p->vertex].distance = p->distance
                a[p->vertex].vertex = i
        p = p->next
    i = FindNextVertex(a)
    if i == NO_VERTEX_FOUND
        break

for i = 1 to Length(a) – 1
    if a[i].vertex != i
        Print (linefeed)
        Print (i)
        Print (“--”)
        Print (a[i].vertex)
```

Additional Function:

```
FindNextVertex (a)
nextVertex = NO_VERTEX_FOUND
shortestDistance = HIGH_VALUE
for i = 1 to Length(a) – 1
    if a[i].distance <= shortestDistance and a[i].known == FALSE
        nextVertex = i
        shortestDistance = a[i].distance
return nextVertex
```

Program Output:

2--1
3--4
4--1
5--7
6--7
7--4

The phase that finds the shortest edges has $O(|V|^2 + |E|)$.

The FindNextVertex function reads all $|V|$ vertices $|V|$ times (i.e. $O(|V|^2)$*).*

For each vertex, each adjacent vertex is checked to determine whether the distance can be reduced. This requires $|E|$ edge traversals (i.e. $O(|E|)$*).*

Complexity-

By using Binary min heap = $O((V+E) \log V)$

By using Fibonacci min heap = $O(E+V \log V)$

By using Binomial min heap = $O(V+E)$

Kruskal's Algorithm

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

Algorithm

1. create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
2. create a set *S* containing all the edges in the graph
3. while *S* is nonempty and *F* is not yet spanning

- remove an edge with minimum weight from *S*
- if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree.

The following code is implemented with a disjoint-set data structure. Here, we represent our forest *F* as a set of edges, and use the disjoint-set data structure to efficiently determine whether two vertices are part of the same tree.

```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F:= F ∪ {(u, v)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Complexity

For a graph with E edges and V vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

- E is at most and .
- Each isolated vertex is a separate component of the minimum spanning forest. If we ignore isolated vertices we obtain $V \leq 2E$, so log V is .

We can achieve this bound as follows:

first sort the edges by weight using a comparison sort in $O(E \log E)$ time;

this allows the step "remove an edge with minimum weight from S" to operate in constant time. Next, we use a disjoint-set data structure to keep track of which vertices are in which components.

We place each vertex into its own disjoint set, which takes O(V) operations.

Finally, in worst case, we need to iterate through all edges, and for each edge we need to do two 'find' operations and possibly one union. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform O(E) operations in $O(E \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

Or

1. Create min heap tree for E- Edges in O(E) times.
2. Delete one by one edge and add to MST, if no cycle in O(Elog E) time,
3. Continue until (v-1) edges are add to MST in (O(E) times)

   Total time complexity = O (E + Elog E) = O(E logV)