Lecture 3: Minimum Spanning Tree

**Lecture - 3**

**MINIMUM SPANNING TREE**

**Spanning Tree**

Given a connected *undirected* graph *G* = (*V*,*E*), ( ) *T VT ET*

=,is a spanning tree if *VT*= *V* ,

*ET* ⊆ *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)

Lecture 3: Minimum Spanning Tree

V1

2

V2

4 10

1

3

V3

5

2 7

V4

V5

8 4

6

V3

V6 V1 2

1

4

1

2

V4

V7

V2

6

V5

V6

1

❖ ***Prim’s Algorithm***

V7

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* , ( ) *v w w w c* distance,,

.distance = min . .

The general strategy

∙ Choose a starting vertex, say 1

*v*and mark it as known.

*The path length from* 1

*vto* 1

*vis 0.*

*Find all vertices adjacent to* 1

*v. These are* 2

*v ,* 3

*v, and* 4

*v .*

*Adjust the distance and vertex for* 2

*v ,* 3

*v, and* 4

*v(e.g. v*2.distance = *c*1,2 = 2*and*

*v* .= *v).*

2 pathVertex 1

∙ Find the unknown vertex with the smallest distance. Select 4

*v*and mark it as known.

*Find all the vertices adjacent to* 4

*v ,* 2

*v. These are* 1

*v ,* 5

*v ,*6

*v, and* 7

*v .*

*v* 3

*vis already known, so no change.*

1

Lecture 3: Minimum Spanning Tree

*Adjust the distance and vertex for* 2

*v ,* 5

*v ,* 6

*v, and* 7

*v* 3

*v* .= *v)*

3 pathVertex 4

*v(e.g. v*3.distance = *c*4,3 = 2*and*

*v , v*2.distance = *c*4,2 = 3*. Since v*2.distance = 2*, then no change to* 2

*For* 2

*v*

∙ Find the unknown vertex with the smallest distance (could be either2 *v*or 3

*v*). Select

*v*and mark it as known.

2

*Find all vertices adjacent to* 2

*v. These are* 1

*v ,* 4

*vand* 5

*v .*

*vand* 4

*vare already known, so no change.*

1

*v . v*5.distance = *c*2,5 =10*. Since v*5.distance = 7*, then*

*Adjust the distance and vertex for* 5

*no change to* 5

*v .*

∙ Find the unknown vertex with the smallest distance. Select 3

*v*and mark it as known.

*Find all the vertices adjacent to* 3

*v ,* 4

*vand* 4

*v . These are* 1

*vand* 6 *v .*

*vare already known, so no change.*

1

*v . v*6.distance = *c*3,6= 5*. Since* .distance 8 *v*6=*, then*

*Adjust the distance and vertex for* 6

.distance 5 *v*6=*and* 6 pathVertex 3

*v* .= *v .*

∙ Find the unknown vertex with the smallest distance. Select 7

*v*and mark it as known.

*Find all the vertices adjacent to* 7

*v. These are* 4

*v ,* 5

*vand* 6

*v .*

*vis known, so no change.*

4

*Adjust the distance and vertex for* 5

*vand* 6

*v .*

*v*5.distance = *c*7,5 = 6 *,* 5 pathVertex 7

*v* .= *v*

*v*6.distance = *c*7,6=1*, v*6.pathVertex = *v*

∙ Find the unknown vertex with the smallest distance. Select 6

*v*and mark it as known.

*Find all the vertices adjacent to* 6

*v. These are* 3

*v ,* 4

*vand* 7

*v .*

*These are all known, so no change.*

∙ Find the unknown vertex with the smallest distance. Select 5

*v*and mark it as known.

*Find all the vertices adjacent to* 5

*v. These are* 2

*v ,* 4

*vand* 7

*v .*

*These are all known, so no change.*

Lecture 3: Minimum Spanning Tree

v known distance path vertex *v*✔ 0 ―

1

*v*🗶

2

*v*🗶

3

*v*🗶

4

*v*🗶

5

*v*🗶

6

*v*🗶

7

v known distance path vertex *v*✔ 0 ―

1

*v*✔ 21

2

*v*

*v*✔ 24

3

*v*

*v*✔ 11

4

*v*

*v*✔ 67

5

*v*

*v*✔ 17

6

*v*

*v*✔ 44

7

Data Structures

*v*

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];

Lecture 3: Minimum Spanning Tree

0 7

vertex

known

distance

firstFollower

vertex

1

1

T 0

4

2 1

T 2

5

3 4 T 2

6

4 1

T 1

7

5 7 T 6

7

6 7 T 1

7

4 T 4

6

distance 10

1 5 4

nextFollower

6 1 1

3

4

4

6

4

4

5

4 3 2 8

7 8 6

2

1

1

5

2 3

4

2

2

4

7

3 2

2 3

1

1

10 4 5

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

Lecture 3: Minimum Spanning Tree

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:*

Lecture 3: Minimum Spanning Tree

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 (*V* + *E* )

2

O .

*The FindNextVertex function reads all Vvertices Vtimes (i.e.*( )2

O *V).*

*For each vertex, each adjacent vertex is checked to determine whether the distance can be reduced. This requires Eedge 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* )

Lecture 3: Minimum Spanning Tree

**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

o remove an edge with minimum weight from *S*

o 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 |
| --- |

Lecture 3: Minimum Spanning Tree

**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* ≤ 2*E*, 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)