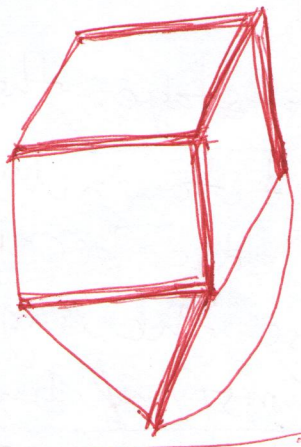
Minimum Spanning Trees

Let $G = (V, E)$ is undirected graph

Spanning Tree in G
- Connected subgraph without cycle.
- It should include all vertices.



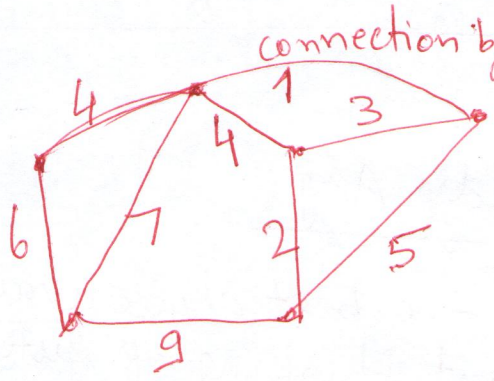
Spanning tree has $|V|-1$ edges.

Minimum Spanning tree:

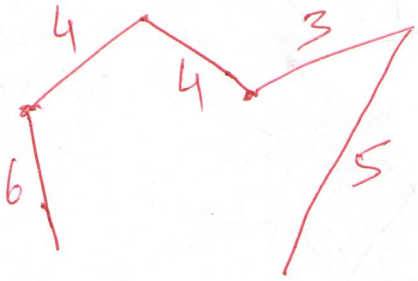
Spanning tree of minimum weight/length

$G = (V, E)$ length $(l): E \to R^+$

or weight $(w): E \to R^+$



Length of the spanning tree = sum of the lengths of the edges in the tree.

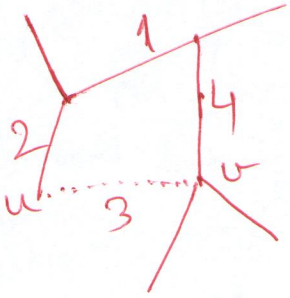


one of the spanning tree

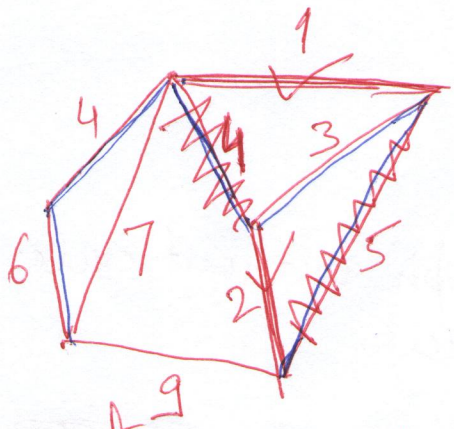
length $= 6+4+4+3+5$

$= 22.$

- Interested in finding the spanning tree having the lowest length.



⇒ In this case edge with length 4 can be dropped and endge with length 3 can be included.

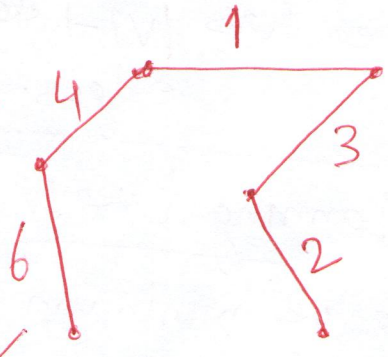


$6+4+4+3+5 = 22$

reduction after dropping the edge with more length and including the edge with lower length.

$22 - 3 - 3 = 16$

Resultant Tree after reduction



⇒ Now at this stage, there is no benefit to include any other edge because they will increase the length.

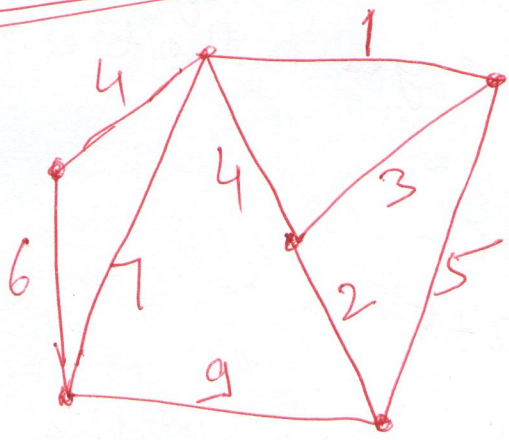
→ This is minimum spanning tree (MST) of length 16.

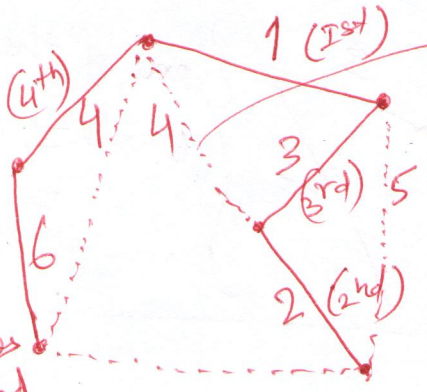
→ This algorithm will be costlier algorithm.

Kruskal's Algorithm for MST.

→ Greedy Algo.

→ Make the best choice available at each step without thinking of future.





→ will form a cycle so excluded.

we can stop → (5th) at this point as all vertices are included

Kruskal's algorithm for MST

1. Sort edges in increasing order of length $(e_1, e_2, e_3, \ldots, e_m)$ s.t. $l(e_i) \leq l(e_{i+1})$
2. $T \leftarrow \phi$
   for $i \leftarrow 1$ to m do
   if $T \cup \{e_i\}$ is a tree Then
   $T \leftarrow T \cup \{e_i\}$
3. return T

Following Concerns

① Why it is correct? Is it possible to have some other MST?

② How to check? This statement is hiding many things.

Proof of correctness of Kruskal's algorithm

Assumption: all edge lengths are distinct ⇒ Same proof can be extended to non-distinct edge lengths also.

Kruskal's ⇒ $g_1 < g_2 < g_3 < \cdots < g_i < \cdots < g_{n-1}$

MST ⇒ $f_1 < f_2 < f_3 < \cdots < f_i < \cdots < f_{n-1}$

Optimum Tree considering best approach

Want to show both sets must be same.

Proof by contradiction

Suppose these sets differ & the first place they differ is i.

↳ means $g_k = f_k$ for $k \in [1, i-1]$ & $g_i \neq f_i$

Case 1: $g_i < f_i \Rightarrow g_i \neq f_k$ for $k \in [1, i-1]$ because $g_k = f_k$ for $k \in [1, i-1]$

& $g_i \neq f_k$ for $k \in [i+1, n-1]$ because $g_i < f_i$

⇒ g_i is not present in optimum solution

So Add g_i to optimum solution (opt tree)

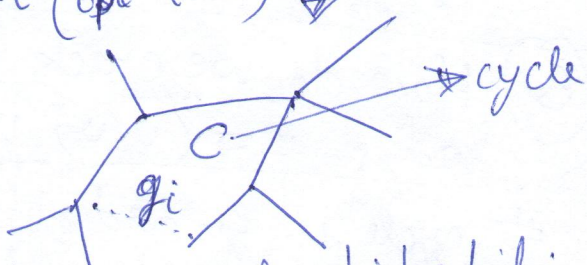
↳ This will form a cycle (C).

Can g_i be the longest edge on the cycle C?

↳ Not possible, thats why kruskal's algo picked it.

Not present in optimum solution → It means, it is longest edge. However,

⇒ Thus it is contradiction.



If g_i is longest, all other edges must be from g_1 to g_{i-1} in cycle = f_1 to f_{i-1}

Case 2: $g_i > f_i$

Kruskal's $g_1 < g_2 < \cdots < g_{i-1} < g_i < \cdots < g_{n-1}$

optimum $f_1 < f_2 < \cdots < f_{i-1} < f_i < \cdots < f_{n-1}$

Case2: $f_i < g_i \Rightarrow f_i \neq g_k$ for $k \in [1, i-1]$

because $f_k = g_k$ for $k \in [1, i-1]$

Why did Kruskal not pick f_i?

& $f_i \neq g_k$ for $k \in [i+1, n-1]$

because $f_i < g_i$

⇓

Because it formed a cycle with $f_i \cup \{g_1, \cdots, g_{i-1}\}$.

$\equiv f_i \cup \{f_1, \cdots, f_{i-1}\}$.

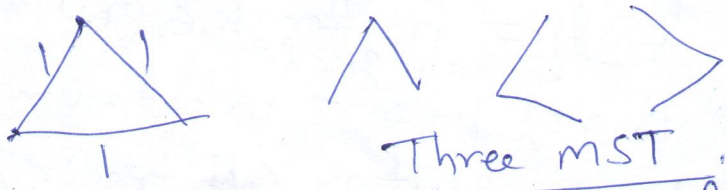
⇒ It means there is cycle in optimum also which is not possible so contradiction.

⇒ It means there is no place where these two sets differ ⇒ So both sets are same.

⇒ Kruskal's algo finds the MST.

Is the MST unique? Yes, if edge lengths are distinct

No, otherwise.

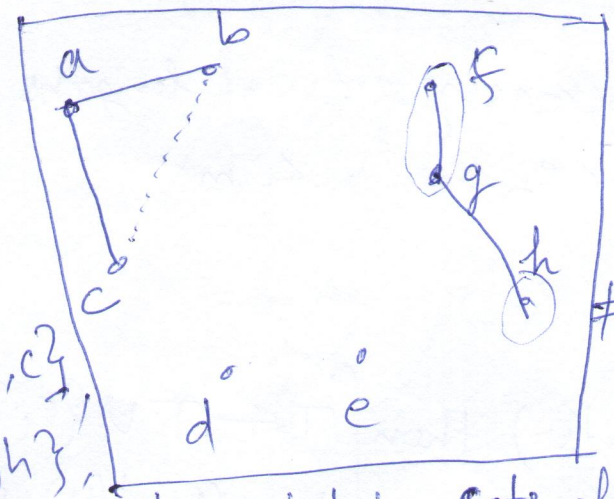


Three MST.

Q) How do we check if a cycle is formed when an edge $e = (u, v)$ is included?

cycle is formed iff u & v are already connected

↳ means u and v are in the same connected components.



whenever an edge is included it combines two connected components into one connected component.

Number of connected component is decreases by one every time a new edge is added.

maintain sets of vertices in each connected component

{a, b, c}, {f, g, h}, {d}, {e}

This data structure needs to be maintained.

whenever, we try to include an edge and both end points of that edge are in the same set, then a cycle will be formed.

The Union Data Structure

Initially n connected components (i.e., all vertices without edges)

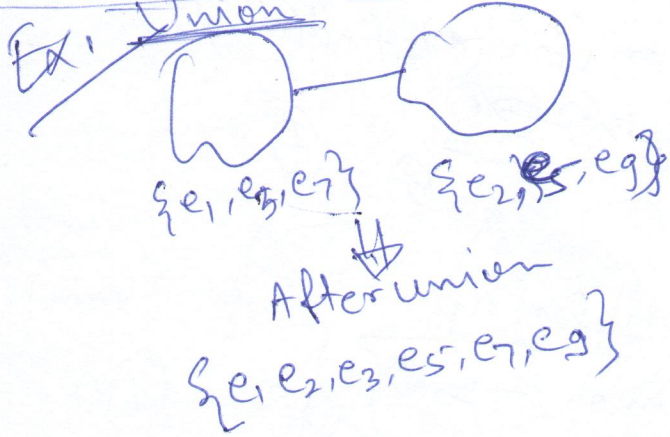
⇓

Eventually we have 1 connected component

$U = \{e_1, e_2, \ldots\ldots e_n\}$ ⇒ Collection of disjoint sets

where, $\{e_1\}, \{e_2\}, \ldots \{e_n\}$ are sets

maintain a collection of disjoint sets under the operations of Union.



Find operation

Find (x, y) returns true if x, y are in the same set.

(or) returns two sets if x, y are not in the same set corresponding to x & y.

Basically ⇒ Find (x, y) should return the sets corresponding to x and y. If both are different, call union.

(6)

Kruskal's algo.

1. sort the edges in increasing order of lengths
$e_1 < e_2 < \dots < e_m$

$T \leftarrow \phi$

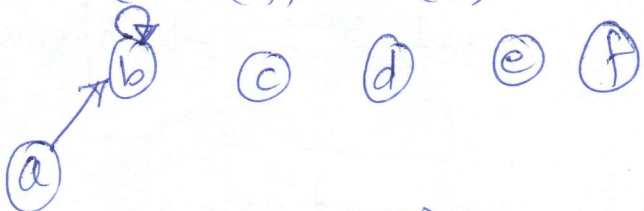
2. for $i = 1$ to m do
   let $e_i = (u, v)$
   if Find$(u) \neq$ Find(v) then $T \leftarrow T \cup \{e_i\}$
   Union(Find(u), Find(v))

Unions number of times: $n-1$
Finds numer of times: $\leq 2m$
If Union & Find take $O(U)$ and $O(F)$ time
then running time $= O(m \log m + U \cdot n + F \cdot m)$

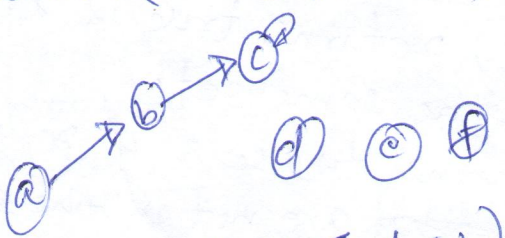
⇒ Needs to find a good data structure to have smaller U & F times.

Union using Tree
$U = \{a, b, c, d, e, f\}$

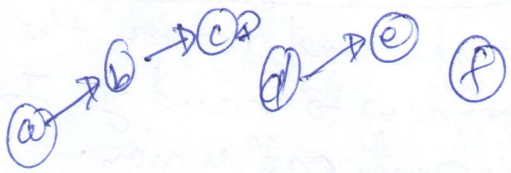
① Union(Find(a), Find(b))

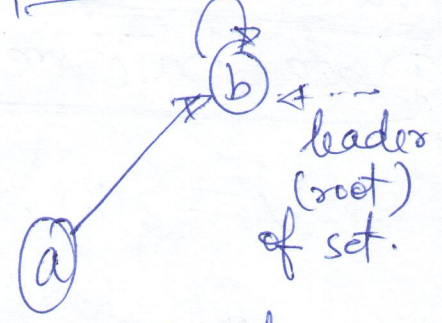


② Union(Find(a), Find(c))



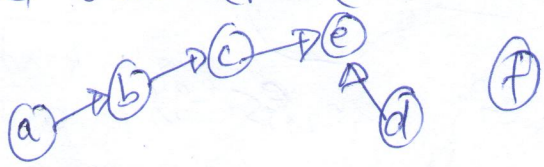
③ Union(Find(d), Find(e))





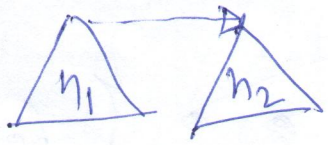
Find will return the reference of root node.

④ Union(Find(a), find(d))



Time complexity
Union → $O(1)$
Find → $O(n)$
still not very good complexity

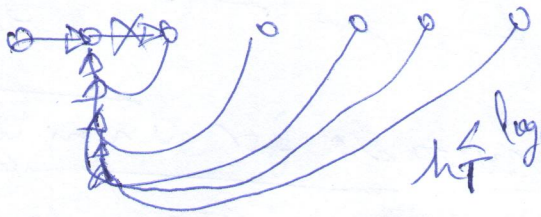
Union by Rank



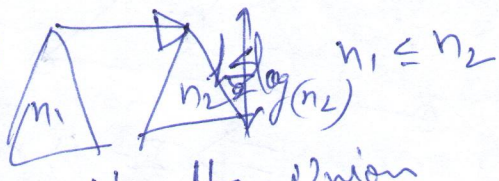
no. of nodes: $n_1 \leq n_2$

claim
A tree with n_1 node has height $\leq \log(n_1)$

Using Induction



$h_T \leq \log(n_1)$

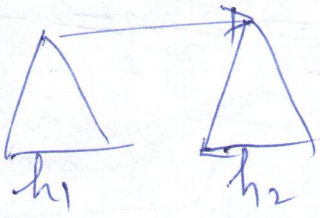


⇒ after Union
no. of vertices $= n_1 + n_2$
need to show height $\leq \log(n_1+n_2)$

If $n_1 \leq n_2$
height of resulting tree $\leq \max(h_2, h_1+1)$
$= (1^{st}\text{ case})\ h_2 \leq \log(n_2) \leq \log_2(n_1+n_2)$
$= (2^{nd}\text{ case})\ h_1+1 \leq (\log_2 n_1 + 1) = \log_2(2n_1)$
$\leq \log_2(n_1+n_2)$

Union by height

It also works.



$h_1 \leq h_2$

claim A tree of height h has at least 2^h nodes.

Proof using induction

$h = \max(h_2, h_1+1)$

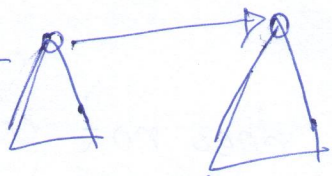
no. of nodes $= n_1 + n_2 \geq 2^{h_1} + 2^{h_2} \geq 2^{h_2}$

$2^{h_1} + 2^{h_2} \geq 2^{h_1} + 2^{h_1} = 2^{h_1+1}$

$\Rightarrow$ no. of nodes $\geq \max(2^{h_2}, 2^{h_1+1}) = 2^{\max(h_2, h_1+1)}$
$= 2^h$

$\Rightarrow$ no. of nodes $(n) \geq 2^h$

Union:
(time) $O(1)$



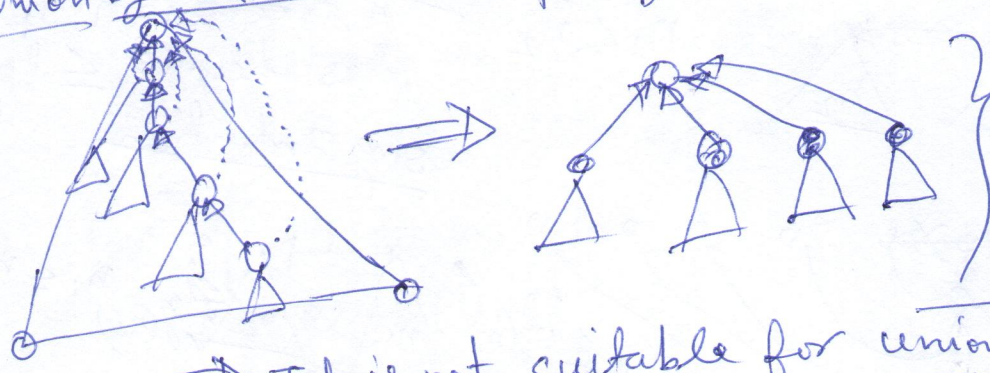
only reference updating

Find:
(time) $O(\log_2 n)$ ⇒ height of the tree

Kruskal's algo time is
$O(m \log m + n + m \log_2 n)$
$= O(m \log_2 n)$

In a connected tree
$\log n \leq \log m \leq 2 \log n$

⑧ Path Compression for Union by Rank → Improving Find time further.



Reduces the height of the tree.

⇒ It is not suitable for union by height.

Prim's Algorithm for MST ⇒

A Cut in graph G is a partition of the vertex set into parts.

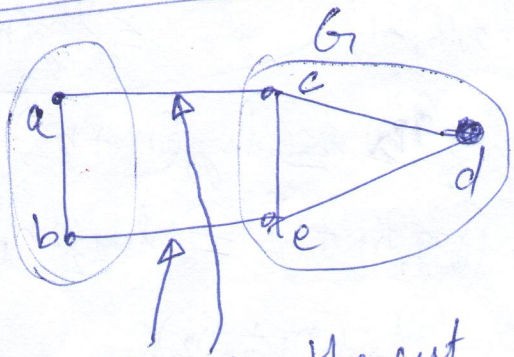
{a,b} {c,d,e}

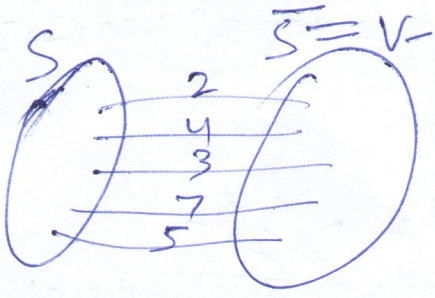
many cut possible

{a,c,d} {b,e}

{c} {a,b,d,e}

Total no. of cuts = $\frac{2^n}{2} - 1 = 2^{n-1} - 1$ (different; ↑ no element in one set)



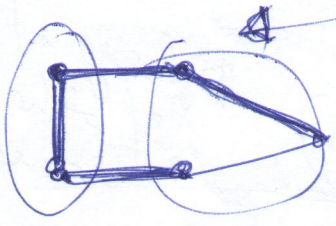


$G = (V, E)$

Assume edge lengths are distinct
↳ for simplification, However same argument would be valid for repetition also.

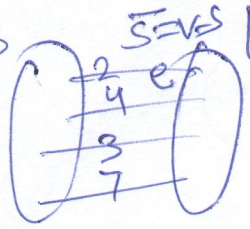
Claim: For any cut $(S, \bar{S})$, the minimum edge in the cut belongs to the MST.

→ However, more than one edges of cut can be also part of MST.

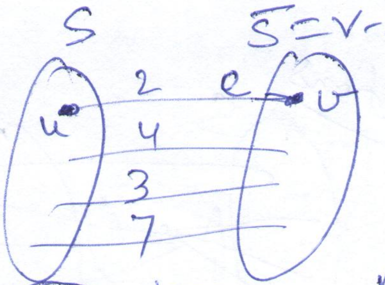


⇒ If the minimum edge of cut is not in MST then that MST is not correct.

Proof by Contradiction:



Let T be a MST which does not contain edge e.



Proof: By Contradiction

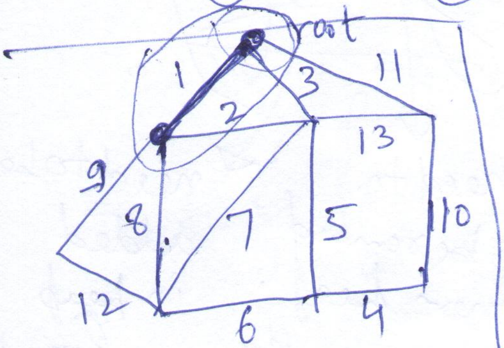
Let T be a MST which does not contain edge e.

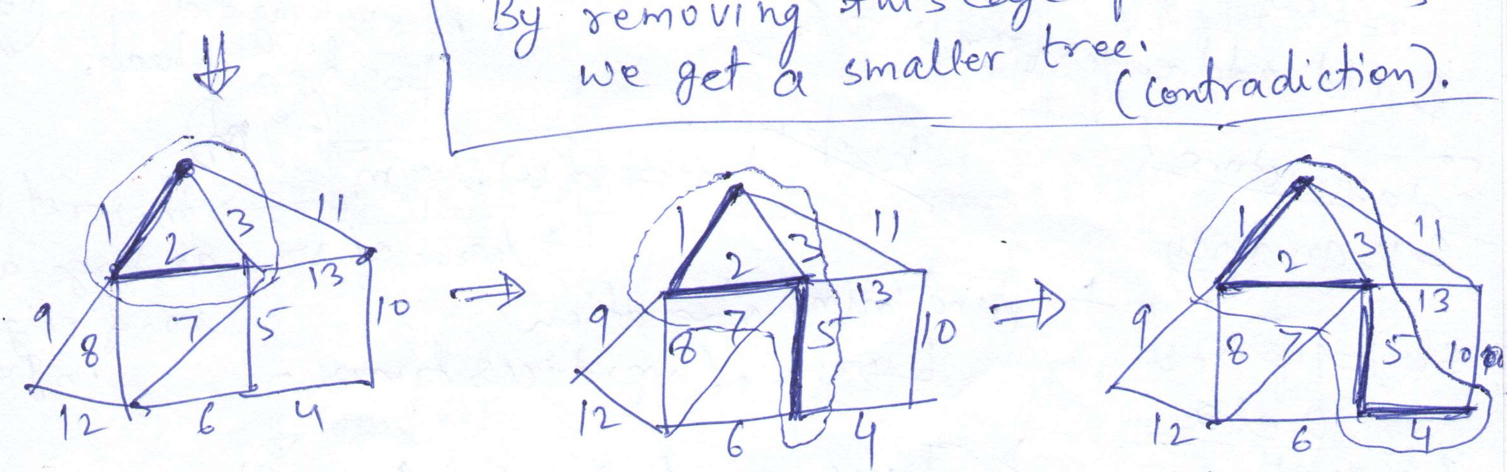
Add e to T
↳ So cycle will be formed, say C.

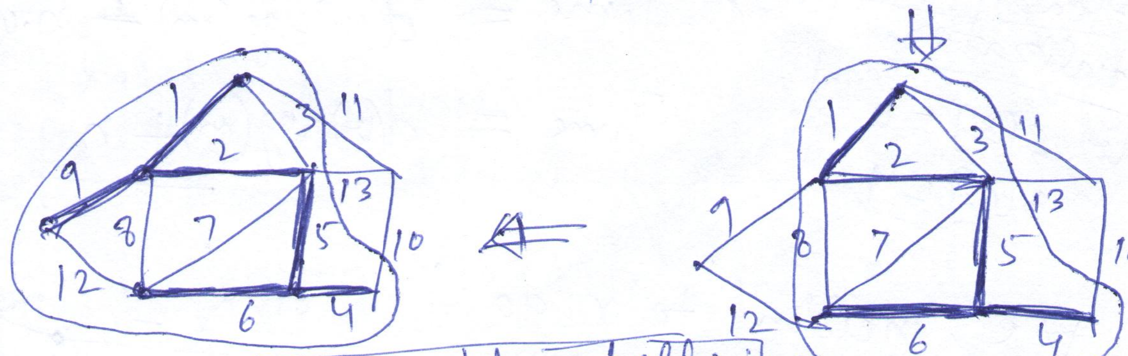
C contains at least one edge (other than e) of the cut.

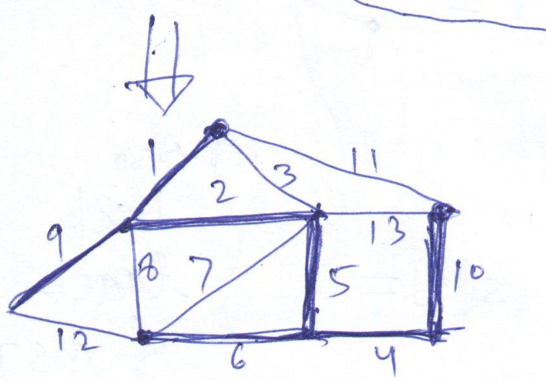
⇒ C contains an edge of length more than the length of e.

By removing this edge from T ∪ {e} we get a smaller tree. (contradiction).

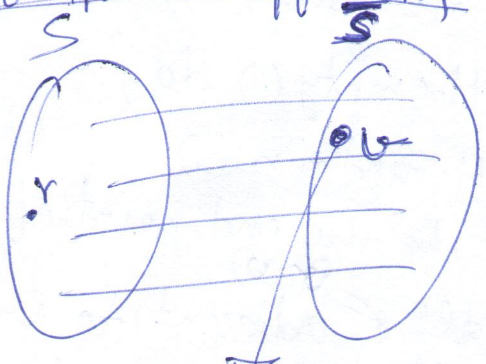






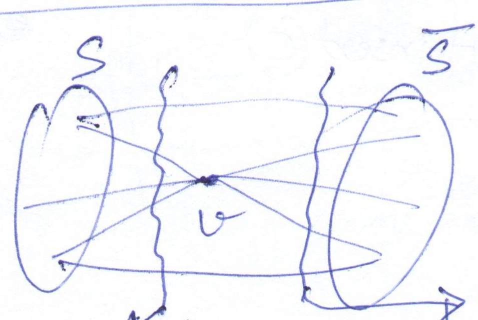


How to implement efficiently



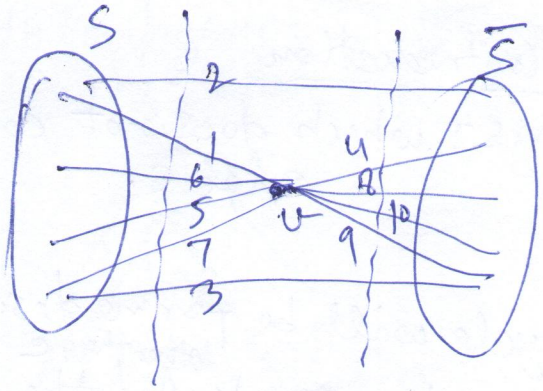
one-bit information is needed to classify any vertex either in set S or $\bar{S}$.

let The vertex going to move from $\bar{S}$ to S.

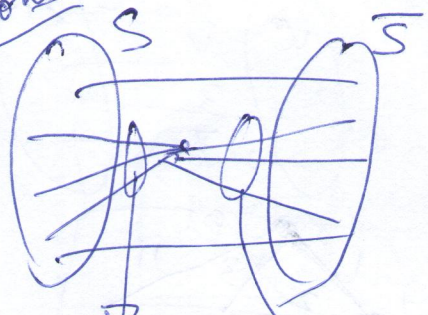


min is known

after v moved we have to find min of these edges.



Heap operations



need to be removed from heap. by knowing the position in heap.

need to be added in heap ($\log m$)

use heap ⇒ will always contain the edges

remove edges from v to S
& add add edges from v to $\bar{S}$

Data structure:
S is an array
$s[v]=1$ if $v \in S$
$=0$ else
& Heap H.
Initialization
$\forall v$ $s[v]=0$
$s[r]=1$

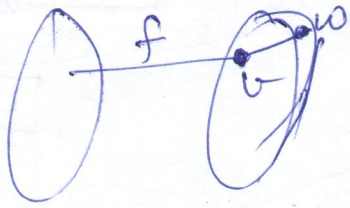
heap: time $= d(v).\log m$ ($\log m$)
↳ degree of v
⇒ in worst case as large as no. of edges in graph.

+ find(min) operation
↳ no. of vertices times

$\Rightarrow$ Time $= d(v)\log(m) + n.O(1)$

Time $= d(v)\log(m) + n$

$\forall e$ incident to r do
H.insert(e)

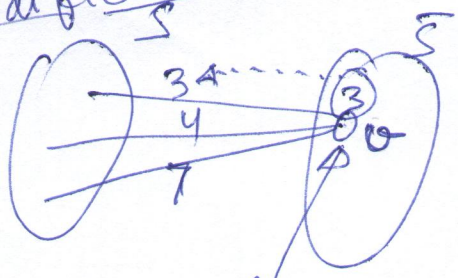


while !H.empty() do{
f = H.findmin()
let v be the end point of f s.t. $s[v]=0$
for all $e=(v,w)$ adjacent to v do
if $s[w]==0$ then H.insert(e)
else H.delete(e)
$s[v]=1$
}

loop executed n-1 times.

Also Keep track of MST edges.

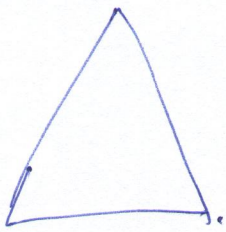
A modification



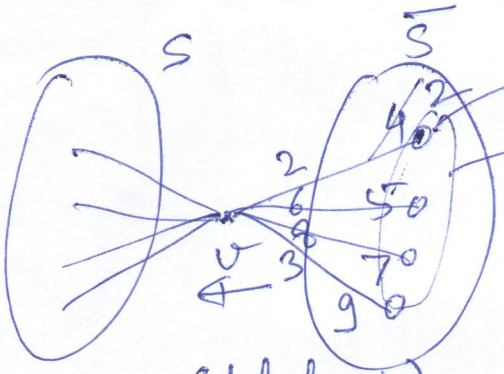
A number 3 is associated with v ← min edge length from S to v.

If there is no edge from S to v then level is ∞.

⇒ heap of vertices



heap has one element for each vertex in $\bar{S}$



level will be decreased to 2. (updated).

the levels need to be updated if v moved from $\bar{S}$ to S

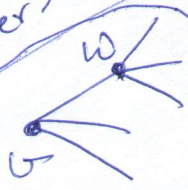
⇒ (deletemin) find min and decrease-priority operations are needed for heap.

~~Initialize~~ S[], label[]

∀ v S[v] = 0, H.insert(v, ∞)

S[r] = 1, H.decreasepriority(r, 0)

better to write H.deletemin() otherwise it will get stuck in infinite loop.




```
while !H.empty() do {
    v = H.findmin()
    for all w adjacent to v do
        if S[w] == 0 then
            if label[w] > length(v,w) then
                label[w] = length(v,w)
                H.decreasepriority(w, label[w])
    S[v] = 1
}
```

running time same complexity as before.