13/4/2020

Spanning tree

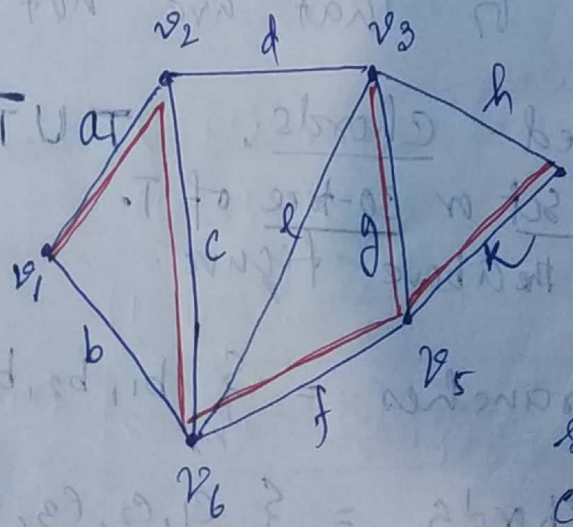
A given graph has many subgraphs.

Obviously some of the subgraphs will be trees.

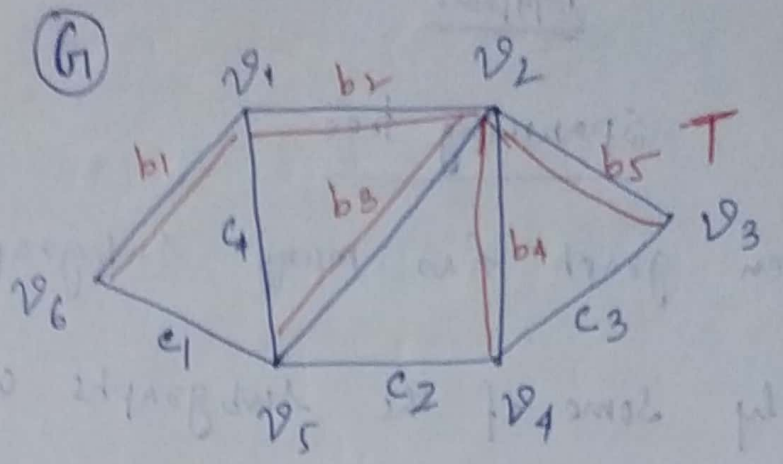
Out of these trees we are interested in special type of trees, called spanning trees.

Defn A tree T is said to be a spanning tree of a connected graph G if T is a subgraph of G and T contains all vertices of G.

ex



Subgraph shown in red color is a valid spanning tree here.



Subgraph shown in Red color is a valid spanning tree here.

So a connected graph may have many spanning trees.

Branch & Chord

For a given connected graph G and its spanning tree T, the edges of T are called branches and edges of G that are not in T are called chords. $T \cup \bar{T} = G$

$\bar{T}$ = chord set or co-tree of T.

ex For the above figure.

Branches = $\{b_1, b_2, b_3, b_4, b_5\}$

Chords = $\{c_1, c_2, c_3, c_4\}$

Rank and Nullity

In a graph there are three fundamental numbers
i) order of a graph
n = Number of vertices in the graph
ii) size of a graph.
e = Number of edges in the graph.
iii) Number of components K of graph

Other two fundamental number of a graph are rank and nullity.

Rank If a graph G has n vertices and K components then each of the components must have at least one vertex.

So from pegion hole principle we can say, $n \geq K$

$\therefore$ $(n-K) \geq 0$ n, K, e are independent

So $(n-K)$ is called rank of the graph. $r = n-K$

if $K=1$ i.e. connected graph.

$\text{rank} = n-K = (n-1)$.

= Number of branches in any spanning tree of the graph.

Nullity

Now a graph with n vertices and K components, each of the component are connected, that means each component must be having edges not less than a tree in the component.

So if K components are having $n_1, n_2, \ldots, n_K$ vertices and $e_1, e_2 \ldots e_K$ edges, than

$$e_1 \geq (n_1-1)$$
$$e_2 \geq (n_2-1)$$
$$\vdots$$
$$e_K \geq (n_K-1)$$

Summing $(e_1+e_2+\cdots e_K) \geq [(n_1+n_2+\cdots+n_K) - K]$

$$e \geq (n-K)$$

$$\boxed{e-n+K \geq 0}$$

(5)

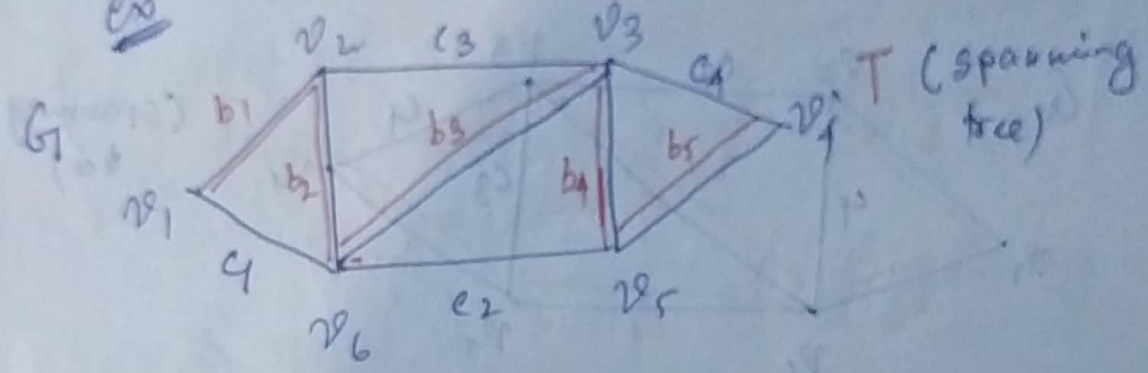
$\mu = e - n + k$ = Nullity of the graph.

if $k = 1$, connected graph.

μ = Nullity = $e - n + 1$

= Number of chords w.r.t any spanning of the graph.

ex



Rank = Number of branches $\{b_1, b_2, b_3, b_4, b_5\}$

= 5

Nullity = Number of chords w.r.t T $\{e_1, e_2, e_3, e_4\}$

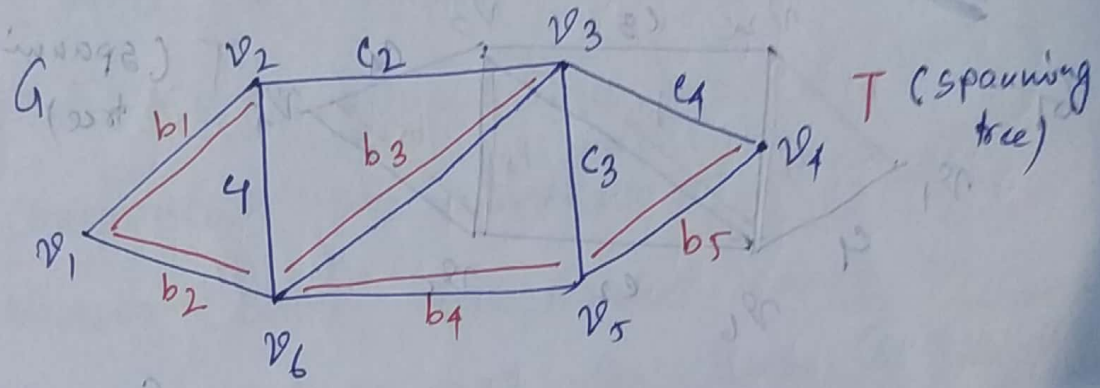
= 4

Rank + Nullity = 5 + 4 = 9 = Number of edges

(6)

Fundamental Circuit.

Let us consider a connected graph G and its any spanning tree T. Adding one chord w.r.t T into T will create exactly one circuit. Such a circuit is called a fundamental circuit.



ex Adding chord c_1 to T will form fundamental circuit

$P_1 = \{ b_1, b_2, c_1 \}$

Similarly for c_2, $P_2 = \{ b_1, b_2, b_3, c_2 \}$

for c_3, $P_3 = \{ b_3, b_4, c_3 \}$

for c_4, $P_4 = \{ b_3, b_4, b_5, c_4 \}$

Theorem: Every connected graph has at least one spanning tree.

If a connected graph G has no circuit it is its own spanning tree.

If G has some circuit:

i) find such a circuit in G

ii) Remove one edge from G. This deletion still leaves graph connected.

iii) Repeat step i) & ii) till there is any circuit left.

The result is obviously a spanning tree of G.

Hence G has at least one spanning tree.

(Proved)

Distance between two spanning trees.

→ Let T_i and T_j are two spanning trees of G.

→ Distance between two spanning trees T_i and T_j of a graph G is defined as the number of edges of G present in one tree but not in the other.

Notation = $d(T_i, T_j)$

$T_i \oplus T_j$ = Ring sum of spanning trees T_i and T_j

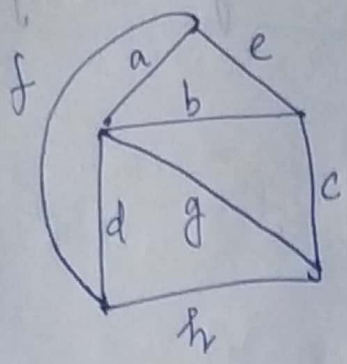
= Subgraph of G containing all edges of G that are either in T_i or in T_j but not in both.

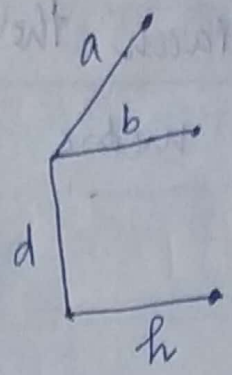
if $\cdot N(g)$ = number of edges in in graph g.

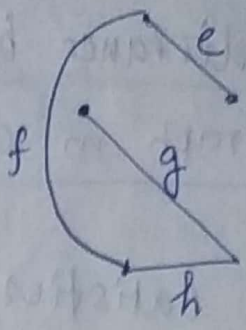
then $\boxed{d(T_i, T_j) = \frac{1}{2} N(T_i \oplus T_j)}$

⑨

ex







G

T_1 {a, b, d, h} T_2 {f, g, h, e}

spanning trees T_1, T_2

Now $d(T_1, T_2)$ = Number of edges of T_2 that are not in T_2

= N {a, b, d}

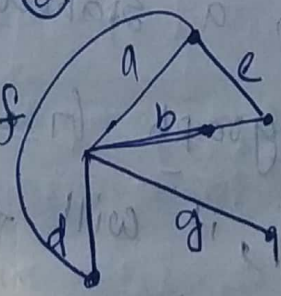
= ③

= Number of edges of T_2 that are not in T_1

= N {e, f, g}

= ③

Now $T_1 \oplus T_2$



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

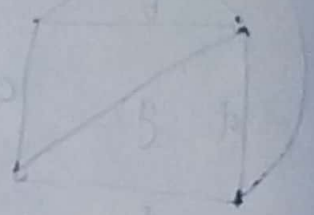
$N(T_1 \oplus T_2)$

So, $d(T_1, T_2) = \frac{1}{2}$

$= \frac{1}{2} \times 6$

$= 3.$

The distance between the spanning trees of a graph is a metric



It satisfies..

a) $d(T_i, T_j) \geq 0$

b) $d(T_i, T_j) = 0$ iff $T_i = T_j$

c) $d(T_i, T_j) \leq d(T_i, T_k) + d(T_k, T_j)$

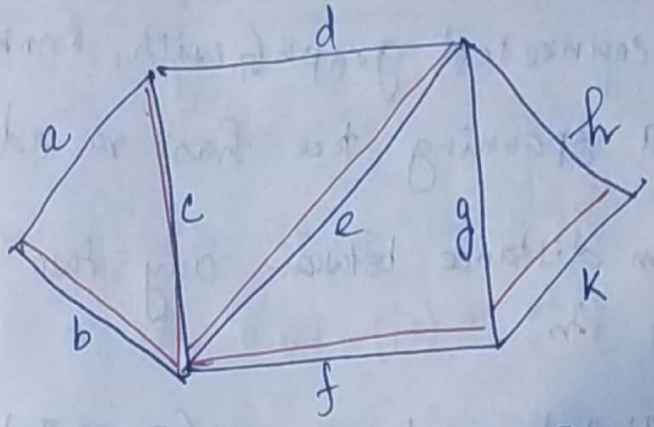
d) $d(T_i, T_j) = d(T_j, T_i)$

cyclic interchange or elementary tree transformation

Let there is a spanning tree T_1 of a connected graph G. If we add a chord to T_1, it will form a fundamental circuit. Now if we remove any branch from this fundamental circuit, it will create a new spanning tree T_2. Generating one spanning tree from another by the above process is called cyclic exchange / interchange.

(11)

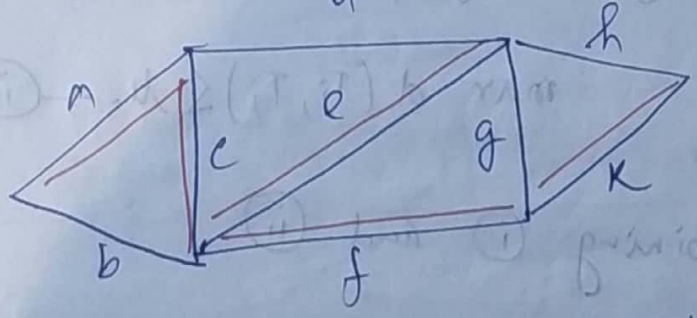
ex.



Let T_1 (spanning tree) has edges

$\{b, c, e, f, K\}$ ⩾

Now Add a (chord) to T_1 and remove branch b from T_1. We get



T_2 has edges $\{a, c, e, f, K\}$

(v) **Maximum distance between two spanning trees of a graph**

In a connected graph G with rank r (i.e. n−1 vertices) a spanning tree has r edges.

So maximum distance between any two spanning trees in G is

$$\max d(T_i, T_j) = \frac{1}{2} \max N(T_i \oplus T_j)$$

$$\leq r, \text{ the rank of } G \quad \text{—(i)}$$

Also if μ is the nullity of G, we know maximum μ chords (edges) can be replaced to get spanning tree T_j from T_i.

Hence $$\max d(T_i, T_j) \leq \mu. \quad \text{—(ii)}$$

So combining (i) and (ii),

$$\boxed{\max d(T_i, T_j) = \min(\mu, r)}$$

Central Tree [best spanning tree to start getting all spanning trees]

For a spanning tree T_0 of a connected graph G, let $\max_i d(T_0, T_i)$ denote the maximal distance between T_0 and any other spanning tree T_i in G.

T_0 is called a central tree of G if

$$\max_i d(T_0, T_i) \leq \max_j (T, T_j)$$

for every tree T of G.

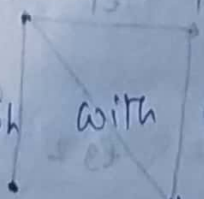
There may be more than one central tree.

Tree Graph

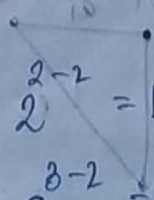
Tree graph is useful for obtaining all spanning trees of G. It is defined as the graph with vertex set of spanning tree T_i and edge set where we connect two spanning trees T' and T iff T' can be obtained from T by one cyclic interchange.

Now we know, starting from any spanning tree we can obtain any spanning tree through cyclic interchange. Hence tree graph is connected.

Counting spanning trees of a connected graph



For a complete graph with n vertices total number of spanning trees is given by Cayley's formulae as:= n^{n-2}.



- So for $K_2 = 2^{2-2} = 1$

$K_3 = 3^{3-2} = 3$

$K_4 = 4^{4-2} = 16$

Spanning tree : Weighted graph

Def^n Let G be a graph and
$W : E(G) \longrightarrow \mathbb{R}$ be a function

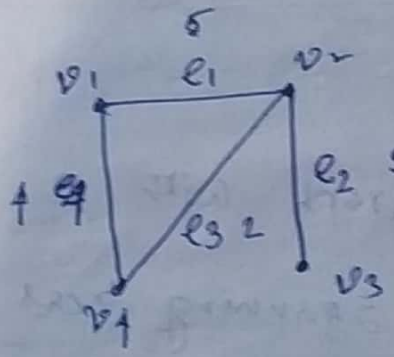
1) The ordered tuple (G, W) is called a weighted graph. The function W is called weight function of (G, W).

2) If G' is a subgraph of G then.

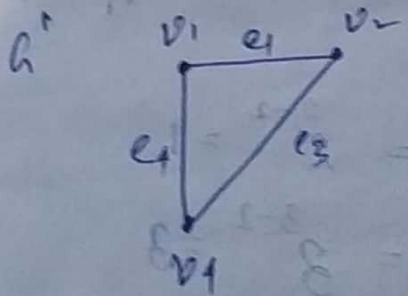
$$W(G') = \sum_{e \in E(G')} W(e)$$

is called weight of G' in G.

Ex^n



$W(e_1) = 5$
$W(e_2) = 3$
$W(e_3) = 2$
$W(e_4) = 4$

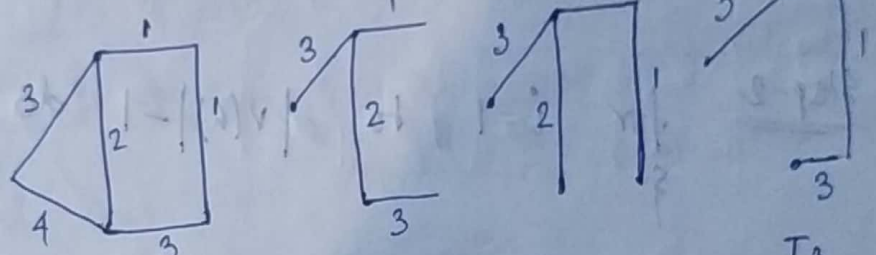


$$W(G') = W(e_1) + W(e_3) + W(e_4)$$
$$= 5 + 2 + 4$$
$$= 11$$

Minimum cost spanning tree:

Now a spanning tree in a connected graph G is a subgraph of G which contains all vertices of G.

Now a connected graph has many spanning trees, each may have different weights. Out of these spanning trees those with minimum weight are of interest in many practical applications [eg: connecting road between cities]



G | T_1 w = 9 | T_2 w = 7 minimum. | T_3 w = 8

Algorithms to find Minimum Spanning trees

a) Kruskal's Algorithm

b) Prim's Algorithm.

Kruskal Algorithm

Input : A connected weighted graph (G, W)

Output : A minimum cost spanning tree T of G.

begin

Step-1 $T_1 = \phi$

Step-2 for $i = 1$; to $|V(G)| - 1$ do

{

Let $e_i \in E(G) \setminus E(T_i)$ be a minimum weight edge and such that $T_i \cup e_i$ is a forest

$T_{i+1} = T_i \cup e_i$

}

Step-3 $T = T_{|V(G)|}$

Kruskal Algorithm

Step-1 In a connected graph G, remove all self loops and remove all parallel edges between any pair of vertices except the edge with least weight.

Step-2 Sort the edges according to non-decreasing order of weights.

Step-3 $T = \emptyset$

Step-4 Select the minimum weight edge and and add it to T such that it doesn't create a cycle.

Step-5 Repeat Step-4 until $|V(G)-1|$ edges are added to T.

How to detect a cycle in an undirected graph ?

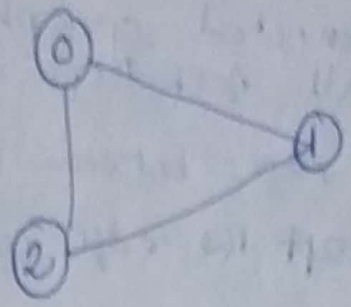
Union-find Algorithm

a) Find : Determine which subset a particular element belongs.

b) Union Join two subset into single set.

(8)

We keep track of the subsets in a 1D array, call it parent.



For each edge, make subsets using both the vertices of the edge. If both are in same subset, a cycle is found.

Initially

V	0	1	2
Parent	0	1	2

a) For edge 0↔1

Find (0) = {0}
Find (1) = {1}
≠

So Union (0,1) make parent of 0 as 1 or vice versa.

V	0	1	2
Parent	1	1	2

b) For edge 1-2

Find (1) = {1}
Find (2) = {2}
≠

So Union (1,2)

V	0	1	2
Parent	1	2	2

(19)

c) for edge 0-2

Find (0) ⇒ Find (1) ⇒ 2

Find (2) = 2.

Hence 0-2 form a cycle here.

PRIM's Algorithm

Step-1 $V_1 = \{u\}$ any $u \in V(G)$ and $V_2 = V(G) - u$, $T = \emptyset$

Step-2

Select any vertex to V_1

a) $V_2 = V_2 \cup u$ Add u to set V_2

b) $V_1 = V_1 - u$ Remove u from V_1

Step-3 Find the minimum weight edge e, where e has end vertices x and y, such that $x \in V_1$ and $y \in V_2$ for $\forall x \in V_1$ and $\forall y \in V_2$

$$\min_{\forall x \in V_1, \forall y \in V_2} w(x,y)$$

Step-4

Add such x to V_2 i.e $V_2 = V_2 \cup x$

Remove x from V_1 i.e $V_1 = V_1 - x$

Add e to T i.e $T = T \cup e$.

Step-4 Repeat step 3 $|V(G)| - 1$ times.

Prim Algorithm

① Create an array Parent[] of size |V| and initialize with NULL

② Create a MinHeap (or priority queue) of size |V|. Let MinHeap be H.

③ Insert all vertices into H such that key values of starting vertex is 0 and key value of other vertices is ∞.

④ While H is not empty

a) u = extractMin(H)

b) for every adjacent v of u

if v is in H

{

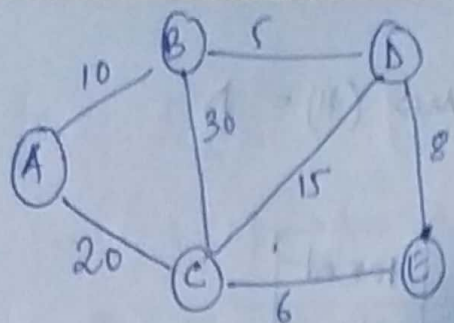
i) update key value of v in H if weight of edge u-v is smaller than current key value of v

ii) Parent[v] = u

}

(21)

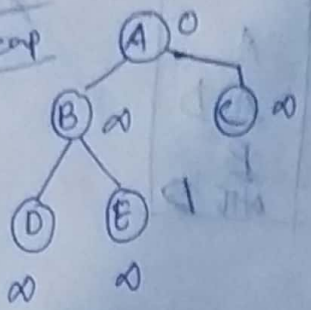
ex



vertex	key	Parent
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

Initially

Min Heap



Step-1

extractMin(H) = A

vertex	key	Parent
A	0	NIL
B	~~∞~~ 10	~~NIL~~ A
C	~~∞~~ 20	~~NIL~~ A
D	∞	NIL
E	∞	NIL

Adjacent vertices

B, C.

Step-2

extractMin(H) = B

Vertex	Key	Parent
A	0	NIL
B	10	A
C	20	A
D	~~∞~~ 5	~~NIL~~ B
E	∞	NIL

Adjacent B

C, D

(22)

Step-3 extractMin(H) = D

vertex	key	parent
A	0	NIL
B	10	A
C	~~20~~ 15	~~A~~ D
→ D	5	B
E	~~∞~~ 8	~~NIL~~ B

Adjacent D

C, E

Step-4 extract Min (H) = E

vertex	key	parent
A	0	NIL
B	10	A
C	~~15~~ 6	~~B~~ E
D	5	B
→ E	8	D

Adjacent

C

Step-5 extractMin (H) = C

vertex	key	Parent
A	0	NIL
B	10	A
→ C	6	E
D	5	B
E	8	D

Adjacent

Ø

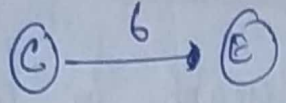
Output

A → Parent (A) → NIL

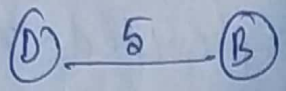
B → Parent (B) → A

A —10— B

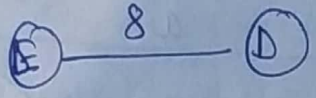
C → Parent (C) → E



D → Parent (D) ⇒ B



E → Parent (E) = D



So finally