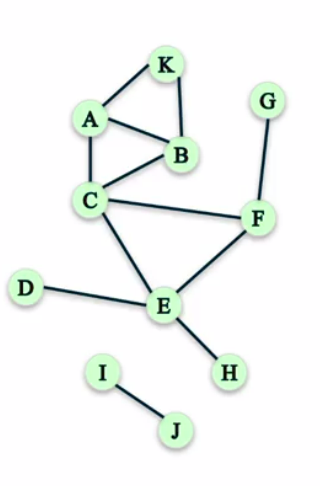
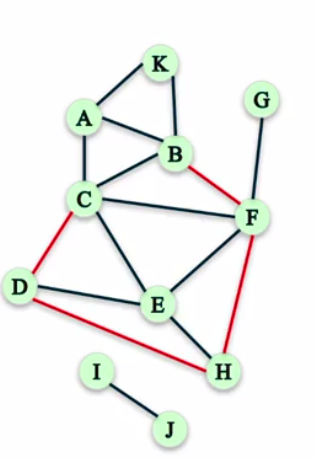
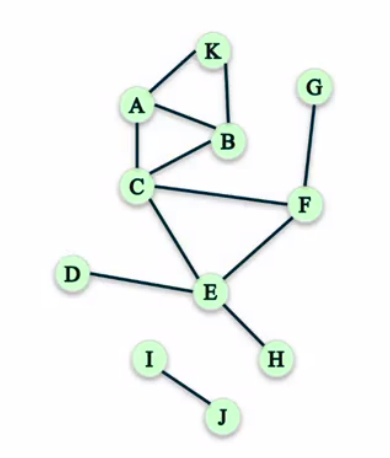
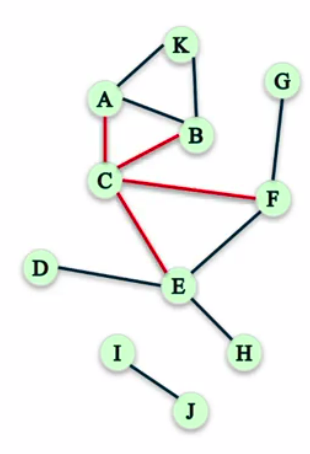
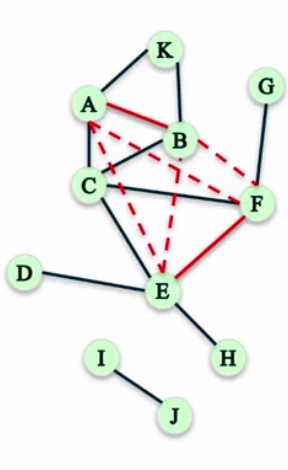
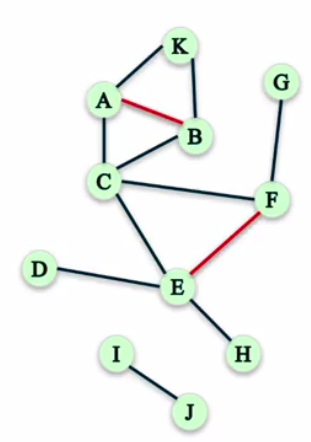
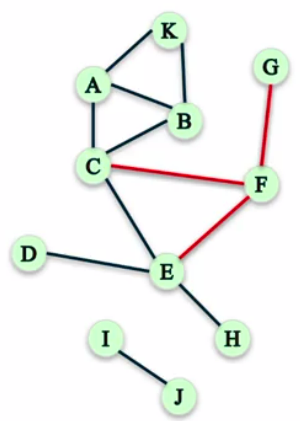
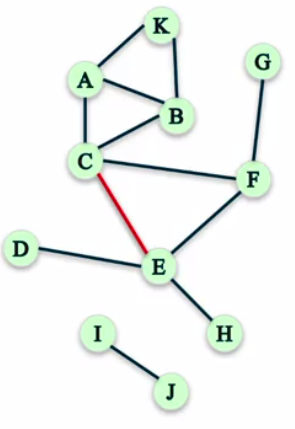
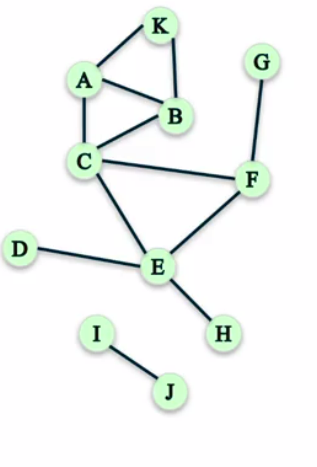
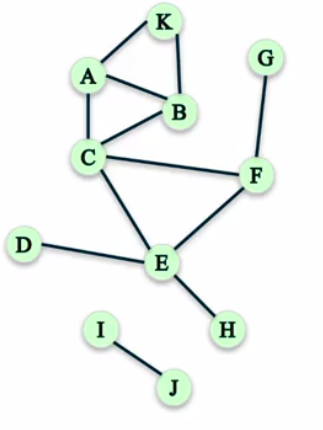
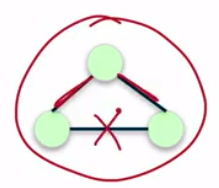
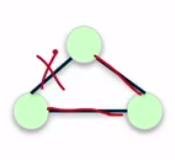
**Clustering Coefficient**  
  
**Triadic Closure**: The tendancy for people who share connections in social   
 network and to form a connection themselves to become connected.It means people who share lots of friends have an increased likelihood of becoming connected themselves.  
Let us say you have a network like this, and see what edges are likely to come to the network next?  
  
  
   
   
  
Triadic Closure would say that those edges that closed triangles are good candidates for edges that may show up next.  
   
   
  
So, here all the red edges form closed triangles, and so, these are good candidates for edges that comes next.However, we don't always have time stamps(***record the time or date of***), or we don't always know the ordering in which the edges come into the network.sometimes, we want to know whether Triadic Closure is present in this network, whether it has lots of triangles or not.So, we'll see with a local version of measuring Clustering.  
  
**Local Clustering Coefficient:**  
The method of measuring Clustering from the point of view of a single node. And, this is called a Local Clustering Coefficient.And, the way it's defined is the fraction of pairs of the nodes friends that are friends with each other.  
   
   
The best way to show how Local Clustering Coefficient works is by showing an example. So, let's say, you wanted to compute the Clustering Coefficient of node C.  
   
**Compute the Clustering Coefficient of node C:**  
Here we should take the ratio of the number of pairs of C's friends who are friends with each other, and the total number of pairs of C's friends.  
  
 #Of pairs of C's friends who are friends  
 #Of pairs of C's friends  
  
C has four friends in the below network. That means that C has a degree of four.The degree is the number of connections that a node has. And, we refer to it as dc as well. So, dc here, which the degree of C is four  
   
   
#Of C's friends=dc=4(the “degree” of C)  
Now, we have to check how many pairs of C's friends are there  
 

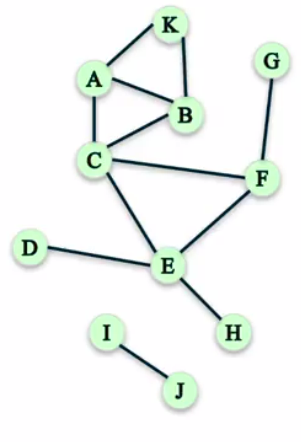
There are four friends of C, and you can easily see that, if you have four pairs of four people, then there are six total possible pairs of people. And so, the total number of pairs of C's friends is six.  
  
 #Of pairs of C's friends= dc( dc-1)  
 2  
 =4(4-1)  
 2  
 =6  
 Now, this is easy to see because there is only four friends of C, but sometimes, there are many more and it might be harder to see how many possible pairs of friends you have. So, what you can do is you can just use this formula here which tells you how many. It's dc times dc-1 over two.  
  
   
   
The number of pairs of friends of C who are friends with each other. Well, there are only two pairs of friends of C that are friends with each other. AB and EF. So, that number is two.  
   
   
 #Of pairs of C's friends who are friends=2  
   
 = 2  
 6  
 = 1  
 3  
So then, the Local Clustering Coefficient of node C is one-third.  
  
**Compute the Clustering Coefficient of node F:**  
  
 #Of pairs of F's friends who are friends  
 #Of pairs of F's friends  
   
F has a degree of three. So, the number of pairs of F's friends is three times two over two which is three  
  
  
   
#Of pairs of F's friends= df( df-1)

2  
 =3(3-1)  
 2  
 = 3  
#Of pairs of C's friends who are friends=1  
  
   
   
There is only one pair of friends of F who are actually friends with each other. That's C and E. And so, the Local Clustering Coefficient of F is also one-third.  
   
The Local Clustering Coefficient of node F= 1  
 3

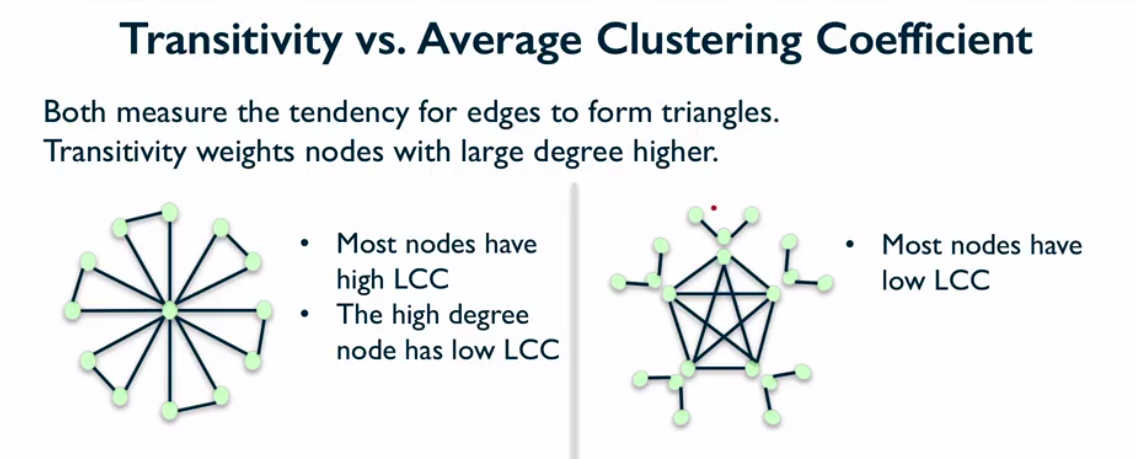
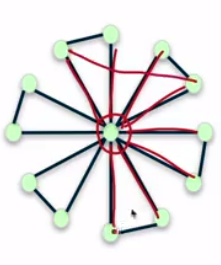
**Compute the Clustering Coefficient of node J:**  
  


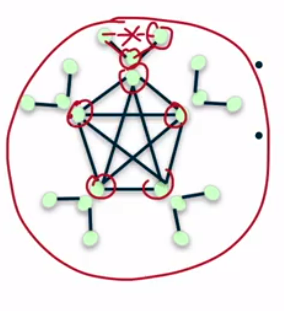
Node J has only one friend which is node I, which means that J actually has zero pairs of friends. And, because that's what we're supposed to put in the denominator  
   
 #Of pairs of J's friends who are friends  
 #Of pairs of J's friends  
   
 #Of pairs of J's friends=0(can not divide by zero)   
   
we're in trouble because we cannot divide by zero. And so, what we're going to do for cases like this, where the definition doesn't work for nodes that have less than two friends,then   
we're going to assume that nodes that have less than two friends have a Local Clustering Coefficient of zero  
So, Local Clustering Coefficient of node J=0  
  
**Local Clustering Coefficient in Network X:**  
  
Let us take a graph in the way and we compute the Clustering. We use the function Clustering to compute the Local Clustering Coefficient of node F.we got 1/3 and also let us check by using network x.  
   
   
import networkx as nx  
G=nx.Graph()  
G.add\_edges\_from([('A','K'),('A','B'),('A','C'),('B','C'),('B','K'),('C','E'),('C','F'),('D','E'),('E','F'),('E','H'),('F','G'),('I','J')])  
  
Input: nx.clustering(G,'F')  
Output: 0.3333333333333333  
  
Input: nx.clustering(G,'A')  
Output: 0.6666666666666666  
  
Input: nx.clustering(G,'J')  
Output: 0.0  
  
For node A, it is 0.66, and for node J, as we had seen, it is zero. So, this allows us to compute the Local Clustering Coefficient of each node in the graph.  
  
**Global Clustering Coefficient:**  
In this we are having two different approaches.  
The first one is to take average Local Clustering Coefficient or all the nodes in the graph. And, you can do this in network X by using the function average Clustering of the graph G.  
  
Measuring clustering of whole network  
  
**Approach 1:A**verage Local Clustering Coefficient or all the nodes in the graph.  
  
Input: nx.average\_clustering(G) (where G is discussed above)  
Output:0.28787878787878785  
  
**Approach 2:** To measure the percentage of “open triads” that are triangles in the network.  
  
   
  
  
  
  
  
  
  
   
   
**Open Triad**: Even though a triad consists of three people, an open form of a relationship can alter because two out of three members in this group can clash.Such type are called as open triads.  
  
if we consider the above triangle here, you will notice that it contains three different open triads. The first open triad considers the three nodes and all the edges, these two edges but not this one.  
  
   
In the second open triad we could consider the three nodes and these two edges but not this one.  
  
   
   
In the third open triad we could consider the three nodes and these two edges but not this one.



So, inside each triangle, there are three different open triads. So, if you go out in the network and count how many triangles it has, and then it counts how many possible open triads it has.  
  
we're going to do for the second approach for measuring Clustering Coefficient, which is actually called as Transitivity.  
  
 Transitivity= 3\*Number of closed triads  
 Number of open triads  
  
You can use network X to get the Transitivity of the network by using the function Transitivity.  
  
By considering the graph(G) we can do transitivity  
G=nx.Graph()  
G.add\_edges\_from([('A','K'),('A','B'),('A','C'),('B','C'),('B','K'),('C','E'),('C','F'),('D','E'),('E','F'),('E','H'),('F','G'),('I','J')])  
   
   
  
Input: nx.transitivity(G)  
Output:0.409090909091  
This network has a Transitivity of 0.41.

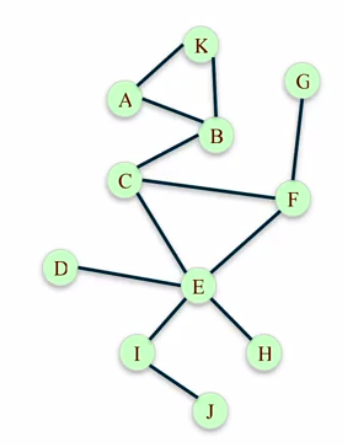
**Difference between Transivity and Average Clustering Coefficient:**

  
  
  
  
if we consider the node inside the wheel, the central node there, that one has a pretty high degree but it has a very low Clustering Coefficient. That is because it has many, many connections and only a few of those are actually connected to each other. But, most of them are not connected.  


For example, the red marked line shows two nodes are not connected,like that if we observe every node they are not connected Even though all of them are friends with that central node. So, in this graph, the average Clustering Coefficient is pretty high.  
  
 So Average clustering coefficient=0.93  
 Transivity=0.23  
  
  
The Transitivity of this network is 0.23. And that's because Transitivity weights the nodes with high degree higher. And so, in this network, there's one node with a very high degree compared to the others.  
  
  
In the second network, most nodes have a very low Local Clustering Coefficient. So, each one of these outer nodes here has a Local Clustering Coefficient of zero because they either have only one friend or they have two friends but those two are not connected.  
 And, there are 15 nodes like that. The nodes inside here, there are only five nodes like that and they have high degree, and then they have high Local Clustering Coefficient. So, when we look at the average Clustering Coefficient and Transitivity.  
  
   
 Average clustering coefficient=0.26  
 Transivity=0.86  
  
So, these two graphs shows the differences between the Local Clustering Coefficient, the average Local Clustering Coefficient, and Transitivity. One weights the nodes with a large degree higher.

**Distance Measures**

**Distance:** The idea here is that sometimes we'd like to know how far nodes are away from each other in a network.Let us see an example.



In the above network let us see how far is node A from node H.To solve this we can use the path.

**Path:**A sequence of nodes connected by an edge

To find the path from A to H first take path-1

**Path-I:**A-B-C-E-H

**Path-II:**A-B-C-F-E-H

Here we are having two paths.So from this we should consider the shortest path.

**Path Length:**The number of steps it contains from beginning to end.So path-1 contains 4 hops and path-2 contains 5 hops

**Distance between two nodes:**The length of shortest path between them.

So here the distance between node A and H is 4.

We can also do this by using networkX

**Input:**nx.shortest\_path(G,'A','H') (G is discussed in previous topic)

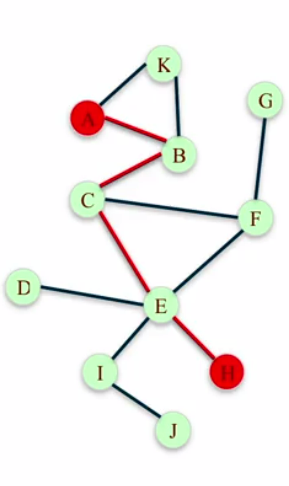
**Output:**['A','B','C','E','H']

It is showing the shortest path between the two noeds A and H

To know the length

**Input:**nx.shortest\_path\_length(G,'A','H')

**Output:**4



Now let us find the distance from node A to every other node.

In such cases we use breadth first search.

**Breadth First Search:**A systematic and efficient procedure for computing distances from a node to all other nodes in a large network.

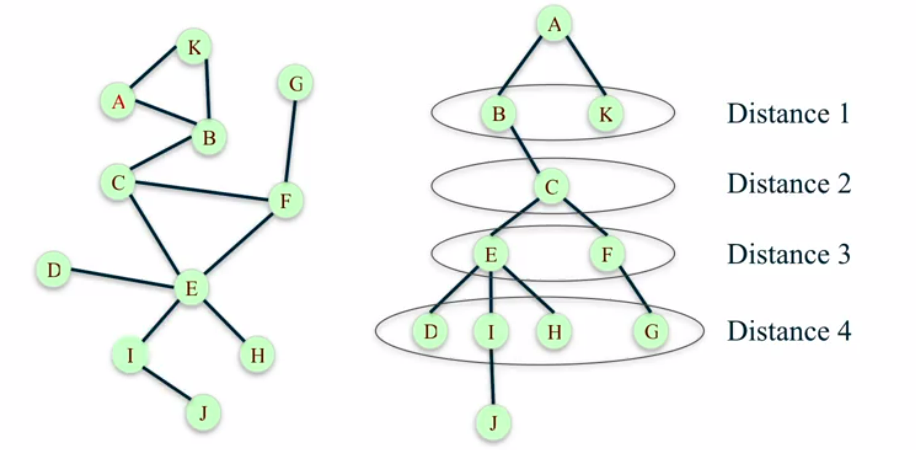
Let us consider the below graph.Check from starting node A

So we start at A and we sort of process the node A by looking at who is connected to A.

In this case, K and B are connected to A and so those are going to be a distance one away because they're the shortest path from each one of those nodes to A it's just one hop.

Now let's say we process node B. Node B is connected to K, A and C. But we've already discovered nodes A and K, so the only node that we discover here is node C.

Now we're going to process node K, and node K is connected to node A and B, but we've already discovered both of those. So the only newly discovered node is node C and it's a distance two away from A.

 Now we process node C which is connected to B, F, and E. And here we've already discovered B so the only two nodes that we discover are F and E and those are a distance three away from A.

Now we're going to process node E.It has five connections and out of those five, C and F we already discovered. So the only new ones are the other three which are D, I and H.

Now we process node F which is connected to three nodes G, C and E. But the only one we haven't discovered yet out of all those is G. now we have to process each one of those newly discovered nodes

Let's process node D which is only connected to E. But we've already discovered E so D does not discover any new nodes.

Now let's go with I. I is connected to E and J. And we haven't discovered J yet, so this one it's assigned to the next layer.

Next we process H which is only connected to E but we already discovered E. And finally, we process G which is connected to F which you've already discovered. So, J is a distance five away.

We have to process J, but J is only connected to I which we already discovered and now we're done. We've processed all the nodes. There are no new nodes to discover.

so here we can see how we efficiently figure out the distance between A and all the other nodes in the network, and this is something that you can program using a computer to do this in an efficient way. You can use networkX to run the breadth-first search algorithm by using the function bfs\_tree.

First let us check the edges by taking T as tree

**Input:**T=nx.bfs\_tree(G,'A')

**Input:**T.edges()

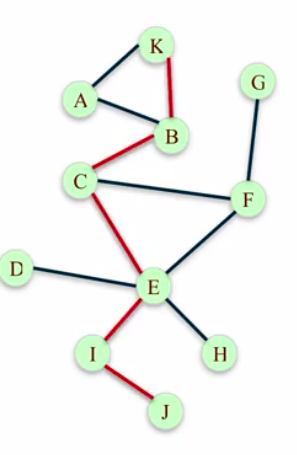
**Output:**[('A','K'),('A','B'),('B','C'),('C','E'),('C','F'),('E','I'),('E','H'),('E','D'),('F','G'),('I','J')]

**Input:**nx.shortest\_path\_length(G,'A')

**Output:**{'A':0,'B':1,'C':2,'D':4,'E':3,'F':3,'G':4,'H':4,'I':4,'J':5,'K':1}

So we got the actual distances between A and all other nodes.

**How to characterize the distance between all pairs of nodes in the graph?**

****

**Average distance** between every pair of nodes

**Input:**nx.average\_shortest\_path\_length(G)

**Output:**2.52727272727

**Diameter:**maximum distance between any pair of nodes.

**Input:**nx.diameter(G)

**Output:**5

**How to summarize the distance between all pair of nodes in the graph?**

**Eccentricity:**The eccentricity of node n is the largest distance between n and all other nodes

We take a node, measure the distance from the node to all the other nodes, and figure out which one of those instances is the largest one of all. In network X, you can use the function eccentricity to get all those distances

**Input:**nx.eccentricity(G)

**Output:**{'A':5,'B':4,'C':3,'D':4,'E':3,'F':3,'G':4,'H':4,'I':4,'J':5,'K':5}

Here A has an eccentricity of five, as we had seen. It has a distance five to some node, which in this case that's J. Because the diameter, the largest possible distance between two nodes was five. But, if we think of a node like for example, node E here, which looks like it's closer to all the other nodes, so that it has a eccentricity of three, which means that no node in this graph is a distance larger than three.

The radius of the graph is the minimum eccentricity.

**Input:**nx.radius(G)

**Output:**3

**Periphery:**The set of nodes that have eccentricity equal to diameter

**Input:**nx.periphery(G)

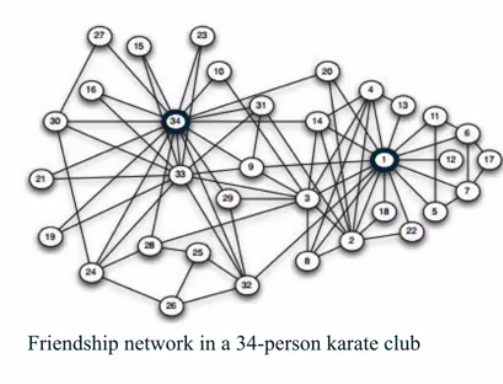
**Output:**['A','K','J']

We can get these set of nodes using networks, network X function periphery. We would sort of imagine these nodes tend to be sort of on the outskirts of the network far away from all the other nodes.

**Center:**The center of the graph is the set of nodes that have eccentricity equal to radius.

**Input:**nx.center(G)

**Output:**['C','E','F']

 Now let us take an example of Karate club network.Here node one is the instructor of the Karate Club, and this node 34 is an assistant, and they have some type of dispute, they're not friends with each other

The club actually splits into two groups, and sort of, this is the separation of the two groups. So one, this set of students on the left go with one of the instructors or with the assistant and the other ones go with the original instructor.

So, if we take this network and apply the definitions about distances that we just covered, we can discover how far nodes are from each other .By using network X we can simply load it by using the function karate club graph.

Input:G=nx.karate\_club\_graph()

Input:G=nx.convert\_node\_labels\_to\_integers(G.first\_label=1)

By using network X we can calculate the remaining

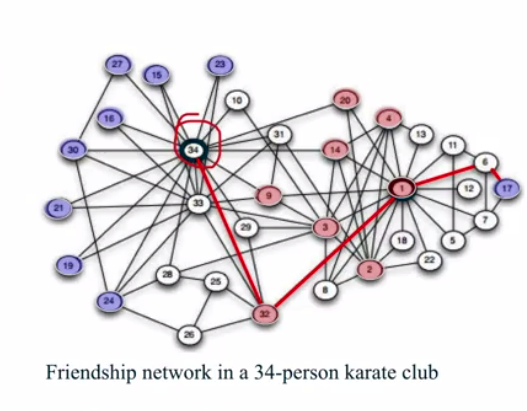
Average shortest path=2.41

Radius=3

Diameter=5

Center=[1,2,3,4,9,14,20,32]

Periphery=[15,16,17,19,21,23,24,27,30]

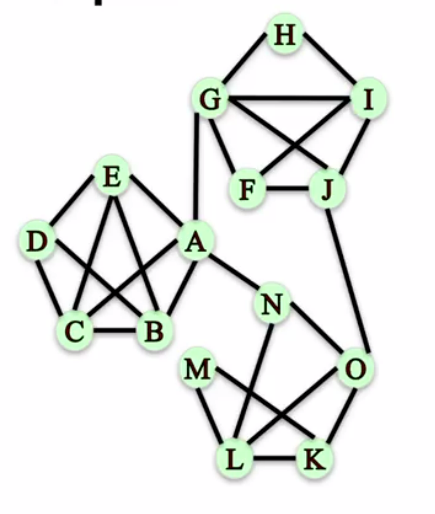


Node 34 looks pretty central however it has distance 4 to node 17

**Connected Components**

**Connected Graph:**First let us see connectivity in undirected graph.The undirected graph is the graph whose edges don't have directions.

An undirected graph is connected if every pair of nodes there is a path between them.



By using network X function

**Input:**G=nx.Graph()

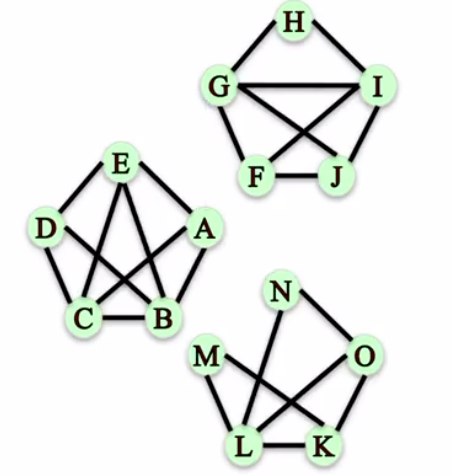
**Input:** G.add\_edges\_from([('A','E'),('A','D'),('A','C'),('A','B'),('A','G'),('A','N'),('G','F'),('G','H'),('G','I'),('G','J'),('J','O'),('O','N'),('O','M'),('O','L'),('O','K')])

**Input:**nx.is\_connected(G)

**Output:**True

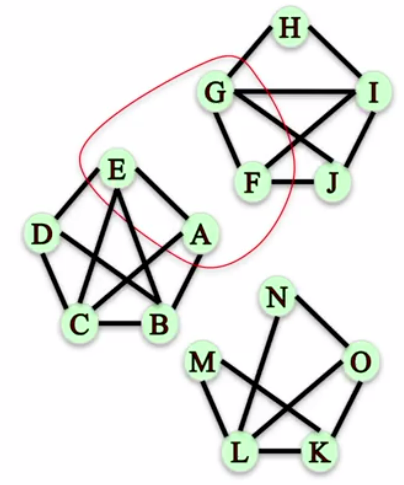
However if we remove edges A-G,A-N and J-O ,the graph becomes disconnected

There is no path between nodes in the three different communities.We cannot find a path to a node in a different community, or in a different set of nodes.



**Graph Components:**

**Connected Component:**A subset of nodes which follows some conditions such as:



i.Every node in the subset has path to every other node.

ii.No other node has path to any node in the subset.

For example, let us consider a graph and check whether is the subset {E,A,G,F} a connected component

Here there is no path between nodes A and

F.

So it does not satisfy the first condition.

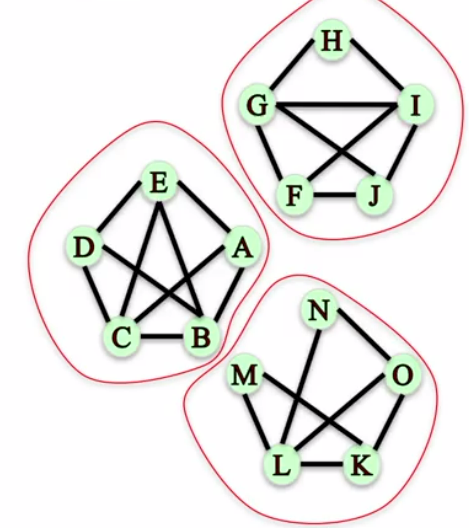
Let us check the second condition.

Is the subset {N,O,K} is the connected component?

By seeing the graph it fails the second condition,because node L has the path to N,O,K.As we discussed in second condition no other node has path to any node in the subset.

So it is not the connected component.

Let us see what are connected components present in the graph.



{A,B,C,D,E},{F,G,H,I,J},{K,L,M,N,O}

These 3 are known as connected components.Let us check by using the network X function.

**Input:**nx.number\_connected\_components(G)

**Output:**3

**Input:**sorted(nx.connected\_components(G))

**Output:**[{'A','B','C','D','E'},{'F','G','H','I','J'},{'K','L','M','N','O'}]

**Input:**nx.node\_connected\_component(G,'M')

**Output:**{'K','L','M','N','O'}

**Connectivity in Directed Graph:**

**A** directed graphs are those that have edges where the edges have direction, so there is a source and a destination.

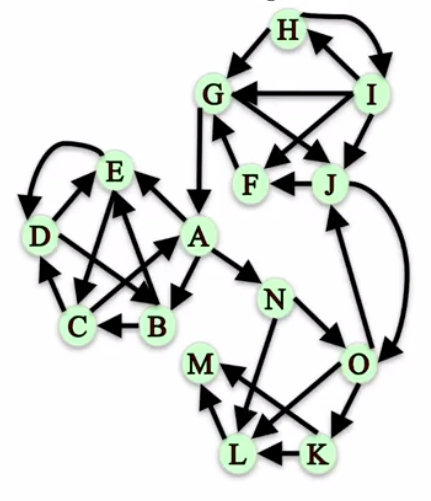
A directed graph has two definitions.

1.Strongly connected

2.Weakly connected

A directed graph is strongly connected if for every pair of nodes say u and v,there is a directed path from u to v and a directed path from v to u.

By seeing the below graph let us check whether it is strongly connected or not by using the network X function.



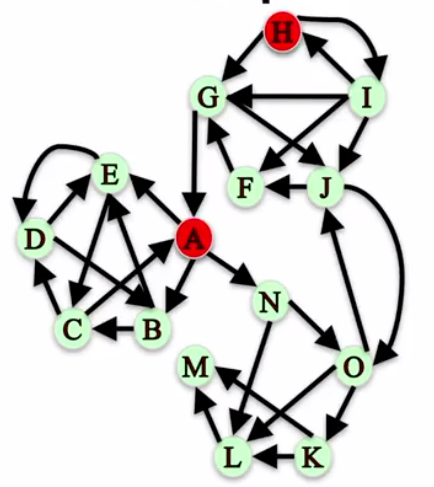
**Input:**G=nx.DiGraph

**Input:**G.add\_edges\_from([('A','E'),('A','B'),('B','C'),('B','E'),('C','D'),('C','A'),('D','B'),('D','E'),('E','C'),('E','D'),('A','N'),('G','A'),('G','J'),('F','G'),('J','F'),('I','J'),('I','G'),('I','H'),('H','I'),('H','G'),('J','O'),('O','J'),('O','K'),('O','L'),('K','L'),('L','M'),('K','M'),('N','L'),('N','O')])

**Input:**nx.is\_strongly\_connected(G)

**Output:**False

It is false because there are many no directed paths between the nodes.For example there is no directed path from A to H

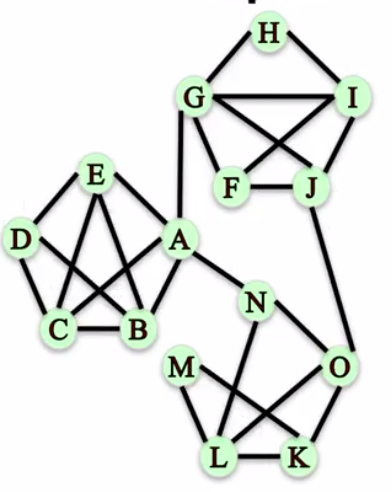


**Weakly Connected:** A directed graph is weakly connected if replacing all directed edges with undirected edges produces a connected undirected graph.

Input:nx.is\_weakly\_connected(G)

Output:True

Network X would say True, this graph is weakly connected because once you turn it into an undirected graph, this undirected graph is connected.



Coming to connected components there are two types

1.Strongly connected component

2.Weakly connected component

**Strongly Connected Component:**A subset of nodes such as

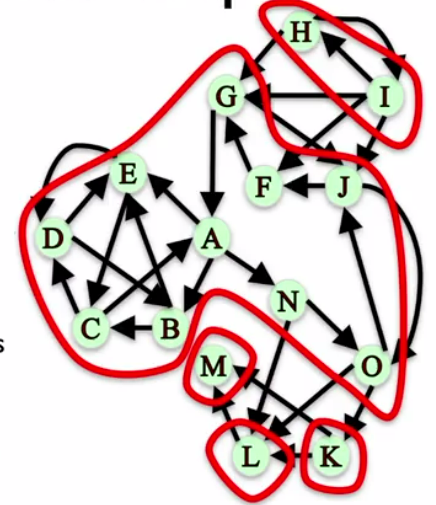
i.Every node in the subset has a directed path to every other node

ii.No other node has a directed path to and from every node in the subset.

Let us check whether the graph is strongly connected component by using network X function

**Input:**sorted(nx.strongly\_connected\_components(G))

**Output:**[{M},{L},{K},{A,B,C,D,E,F,G,J,N,O},{H,I}]



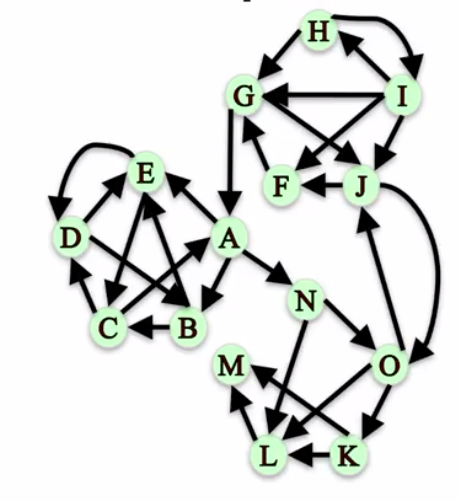
**Weakly Connected Component:**The connected components of a graph after replacing all directed edges with undirected edges.

Let us check by using network X function

**Input:**sorted(nx.weakly\_connected\_components(G))

**Output:**[{'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O'}]

Since the graph is weakly connected it has only one weakly connected component.



**Connectivity And Robustness In Networks**

**Network Robustness:**The ability of a network to maintain it general structural properties when it faces failures or attacks.

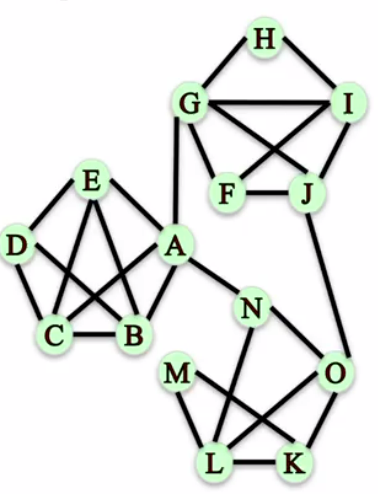
**Type of attacks:**Removal of nodes or edges

**Structural Properties:**Connectivity

**Examples:**Airport closures,Internet router failures,power line failures.

**Disconnecting a graph:**

Let us take an undirected graph and check the node connectivity using network X



**Input:**nx.node\_connectivity(G\_un)

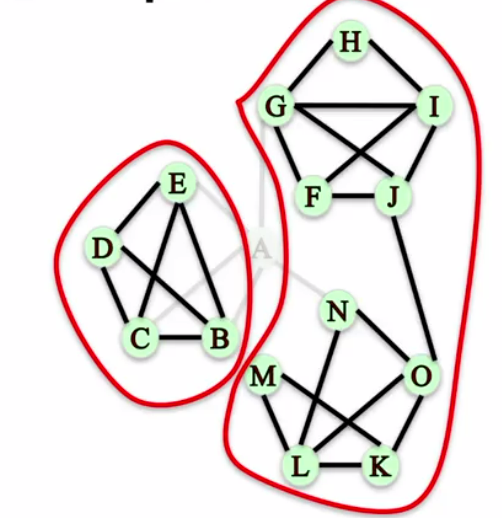
**Output:**1

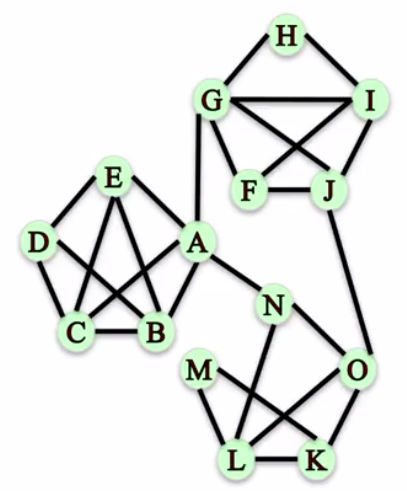
It is showing a single node,so we can check which node it is.

**Input:**nx.minimum\_node\_cut(G\_un)

**Output:**{'A'}

So if we remove node A then the graph goes from connected to being disconnected.





Now check with edges which are smaller and can be removed from the graph in order to diconnect it.

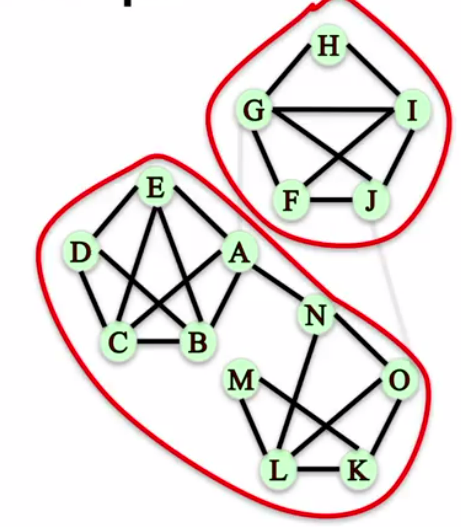
**Input:**nx.edge\_connectivity(G\_un)

**Output:**2

Let us see what are the 2 edges

Input:nx.minimum\_edge\_cut(G\_un)

Output:{('A','G'),('O','J')}

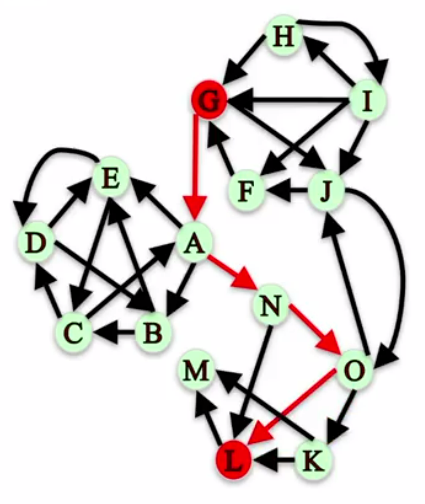


If we remove those two edges it forms like two subsets.

Robust networks have large minimum nodes and edge cuts.

**Simple Paths:**

Imagine node G wants to send a message to node L by passing it along to other nodes in the network.



First we should see the options does G have to deliver the message to L.

Here we can check all simple paths by using source(G) and destination(L)

**Input:**sorted(nx.all\_simple\_paths(G,'G','L')

**Output:**[['G','A','N','L'],

['G','A','N','O','K','L'],

['G','A','N','O','L'],

['G','J','O','K','L'],

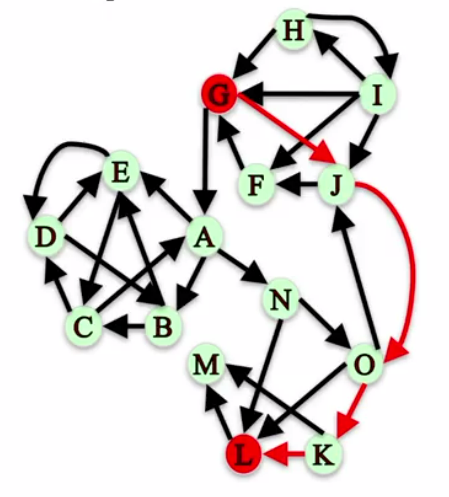
['G','J','O','L']

**Node Connectivity:**If the attacker wanted to block the message from G to L by removing nodes from the network.How many nodes that the attacker should remove?.It can be checked by using the node connectivity.

**Input:**nx.node\_connectivity(G,'G','L')

**Output:**2

To check what are the two nodes again we can use minimum node cut.

 **Input:**nx.minimum\_node\_cut(G,'G','L')

**Output:**{'N','O'}

If we only remove node N message can go on path G->J->O->K>L

If we only remove node O message can go on path G->A->N->L

So there is no possibility to remove one of these two nodes.We should remove the two nodes to block the message.

**Edge Connectivity:**

If we want to block the message from G to L by removing the edges from the network,how many edges can be removed

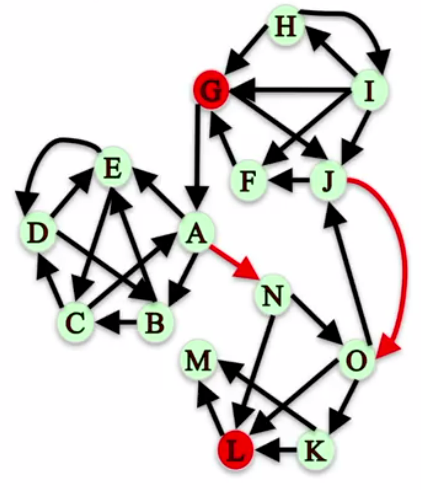
**Input:**nx.edge\_connectivity(G,'G','L')

**Output**:2

To know which edges

Input:nx.minimum\_edge\_cut(G,'G','L')

Output:{('A','N'),('J','O')}



We need to remove the edges A->N and J->O to block the messages from G to L.