MINING SOCIAL-NETWORK GRAPHS

Pranav Singh
Abhinava Mishra
Bhupesh Kumar

INTRODUCTION

- As part of our project, we will be analyzing the large-scale data that is derived from social networks, for example, the “friends” relation found on sites like Facebook.

- An important question about a social network is to identify “communities,” that is, subsets of the nodes (people or other entities that form the network) with unusually strong connections.

- Why study clustering algorithms again? Because, communities almost never partition the set of nodes in a network. Rather, communities usually overlap. For example, you may belong to several communities of friends or classmates.

- We shall cover techniques like **Calculating Betweenness**, **Counting Triangles** and **Finding complete bipartite graphs**, for analyzing such networks.

CHARACTERISTICS OF A SOCIAL NETWORK

- There is a **collection of entities** that participate in the network. Typically, these entities are people, but they could be something else entirely.

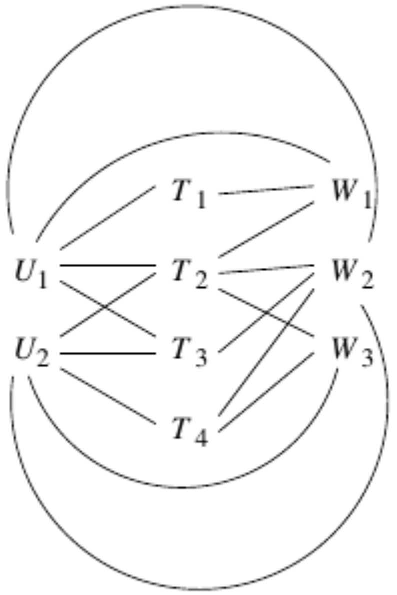
- There is **at least one relationship between entities** of the network. On Facebook or its ilk, this relationship is called friends. Sometimes the relationship is all-or-nothing; two people are either friends or they are not. However, in other examples of social networks, the relationship has a degree as in Google+.

- There is an assumption of **non-randomness or locality**. This condition is the hardest to formalize, but the intuition is that relationships tend to cluster. That is, if entity A is related to both B and C, then there is a higher probability than average that B and C are related.

GRAPHS WITH SEVERAL NODE TYPES

Users at a site like deli.cio.us place tags on Web pages. There are thus three different kinds of entities: **users, tags, and pages**. We might think that users were somehow connected if they tended to use the same tags frequently, or if they tended to tag the same pages. Similarly, tags could be considered related if they appeared on the same pages or were used by the same users, and pages could be considered similar if they had many of the same tags or were tagged by many of the same users.

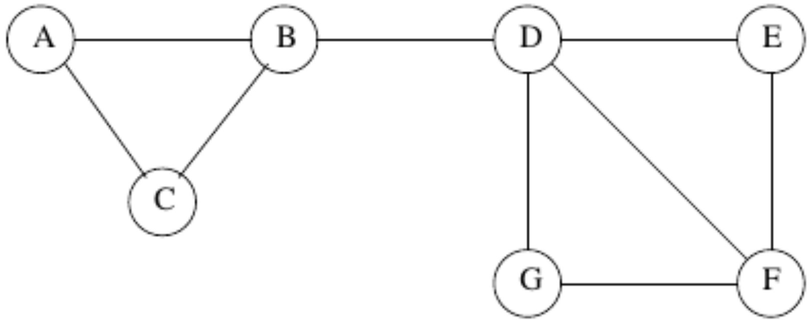
The natural way to represent such information is as a **k-partite graph** for some k > 1. In general, a k-partite graph consists of k disjoint sets of nodes, with no edges between nodes of the same set.



GIRVAN-NEWMAN ALGORITHM

Betweenness

The betweenness of an edge (a, b) is the number of pairs of nodes x and y such that the edge (a, b) lies on the shortest path between x and y. To be more precise, since there can be several shortest paths between x and y, edge (a, b) is credited with the fraction of those shortest paths that include the edge (a, b). As in golf, a **high score is bad**. It suggests that the edge (a, b) runs between two different communities; that is, a and b do not belong to the same community.



CONTD...

The Girvan-Newman algorithm basically involves 3 steps, **to be performed for every node** of the graph by taking that node as the root -

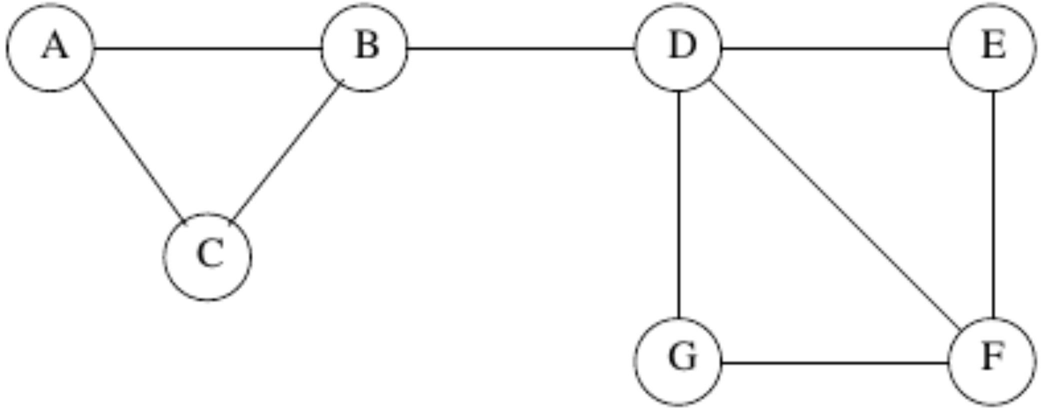
- The algorithm begins by performing a **breadth-first search (BFS) of the graph**, starting at the node X. Note that the level of each node in the BFS presentation is the length of the shortest path from X to that node. Thus, the edges that go between nodes at the same level can never be part of a shortest path from X.

- The second step of the GN algorithm is to label each node by the **number of shortest paths** that reach it from the root. Start by labeling the root 1. Then, from the top down, label each node Y by the sum of the labels of its parents.

- The third and final step is to **calculate credits for each edge e**, that is, the sum over all nodes Y of the fraction of shortest paths from the root X to Y that go through e.

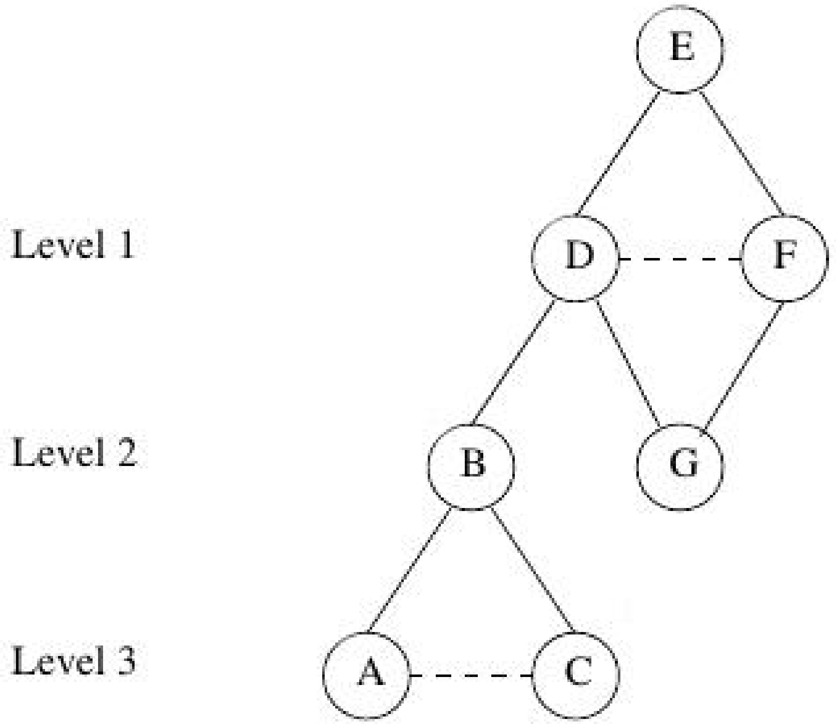
```
Label(X) = Sum of Label of Parents
Credit of nodes(other than leaves) = 1+∑Credit of edges from below
Credit(e) = Credit(C) * (Label(P)/Label(C)) e=edge, C is child P is parent
```

AN EXAMPLE



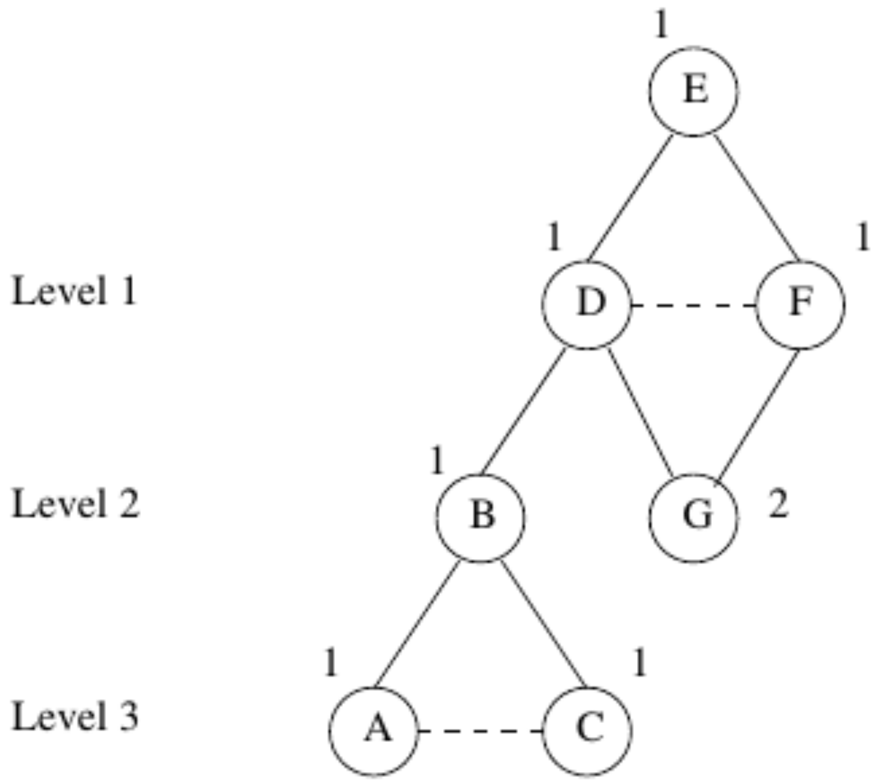
The graph under consideration

Step 1
Perform BFS on the graph with node E as root.

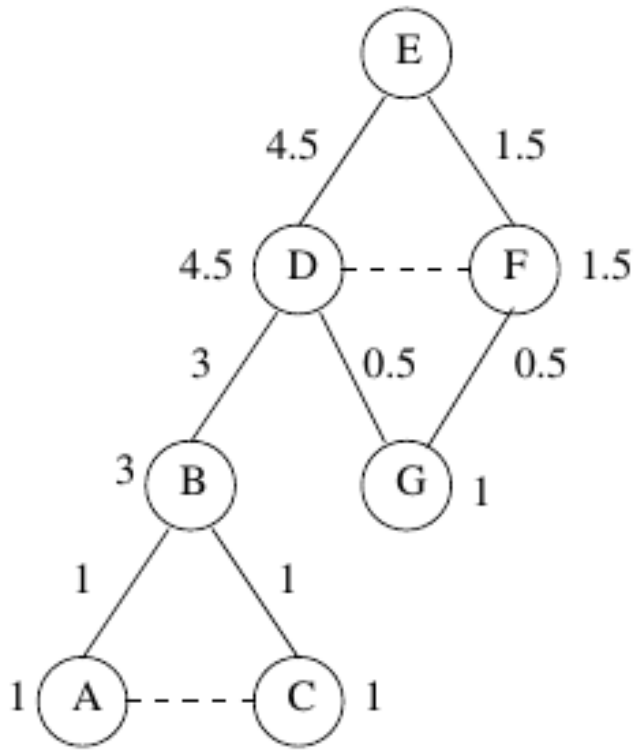


Step 2

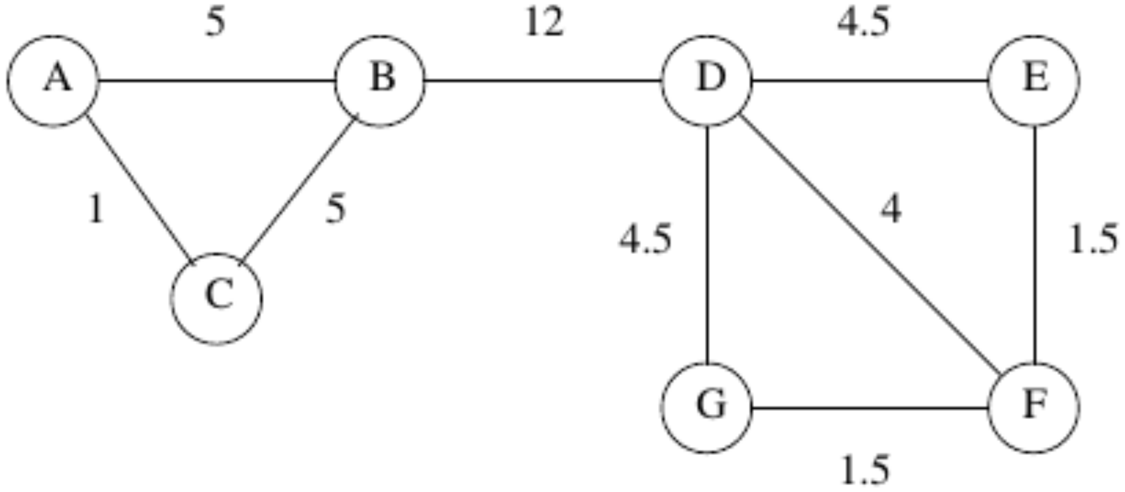
Label the nodes with the number of shortest paths.



Step 3
Calculate credits for each edge

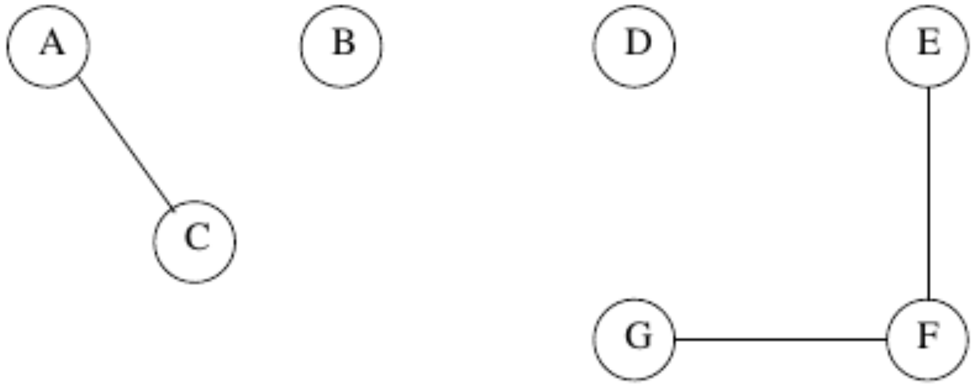


Now, repeat the same process by taking each node as root, add up the betweenness scores for each edge and finally, divide the score by 2 to get the actual value



Final Betweenness scores

The betweenness scores for the edges of a graph behave something like a distance measure on the nodes of the graph. It is not exactly a distance measure, because it is not defined for pairs of nodes that are unconnected by an edge, and might not satisfy the triangle inequality even when defined. However, we can build communities through a **process of edge removal**. Start with the graph and all its edges; then remove edges with the highest betweenness, until the graph has broken into a suitable number of connected components.



PSEUDOCODE

```
foreach node V in graph G do
      Level(V)=0
      Label(V)=1
      Enqueue V in Queue

      while Queue is NOT empty do
            V = Dequeue()
            Push V in Stack
            foreach child C of V do
                  if C is NOT Enqueued before then
                  Enqueue C in Queue
                  Level(C)=Level(V)+1

            Append V as parent of C
            foreach parent V' of V do
                  Label(V)+=Label(V')
```

CONTD...

```
     **while** Stack is NOT empty **do**
          V = Pop()
          **if** V is a leaf **then**
               Credit(V)=1
          **else**
               **foreach** edge e between child C of V **do**
                    Credit(V) += Credit(e)
               Credit(V) += 1

          **foreach** edge e between parent V' of V **do**
               Credit(e) = Credit(V) * (Label(V')/Label(V))

**foreach** edge e in graph G **do**
    Credit(e) = Credit(e)/2
```

MapReduce

Mapper

```
Input: graph G   // preprocessed list

foreach node N in graph G do
     List(edge, credit) = GirvanNewman(N)  // List is a hash which maps to edge to credit

foreach (edge, credit) in List do
      emit <edge; credit>
```

Reducer

```
Input: <edge; credit>
sum = {}

foreach (edge,credit) in Input do
     sum[edge] += credit

foreach (edge,sumOfCredits) in sum do
     emit <edge; sumOfCredits/2>
```

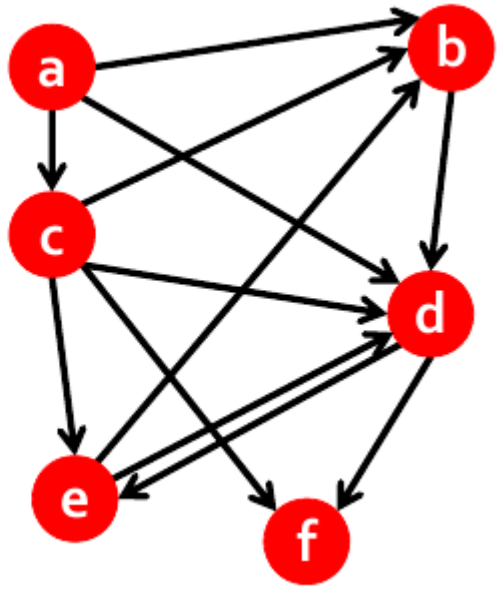
DIRECT DISCOVERY OF COMMUNITIES

In the previous section we searched for communities by partitioning all the individuals in a social network. While this approach is relatively efficient, it does have several limitations. It is not possible to place an individual in two different communities, and everyone is assigned to a community. In this section, we shall see a technique for discovering communities directly by looking for subsets of the nodes that have a **relatively large number of edges** among them.

Finding Cliques - The Intuitive Method

Our first thought about how we could find sets of nodes with many edges between them is to start by finding a large clique (a set of nodes with edges between any two of them). However, that task is not easy. Not only is **finding maximal cliques NP-complete**, but it is among the hardest of the NP-complete problems in the sense that even approximating the maximal clique is hard. Further, it is possible to have a set of nodes with almost all edges between them, and yet have only relatively small cliques.

FINDING COMPLETE BIPARTITE SUBGRAPHS



Support = 2

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

First Pass

a	0
b	3
c	1
d	4
e	2
f	2

Hence, L(1) is {b,d,e,f}

Second Pass

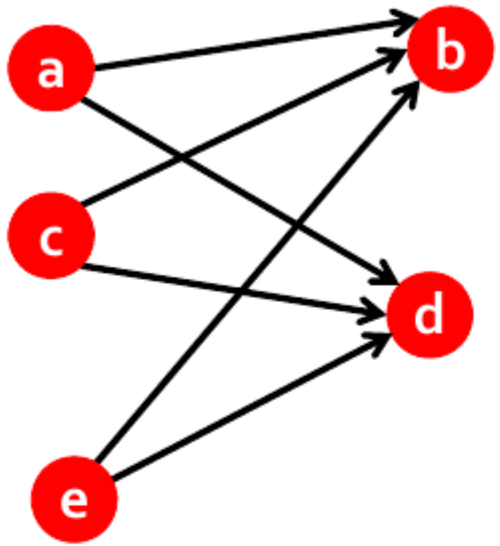
(b,d)	3
(b,e)	1
(b,f)	1
(d,e)	1
(d,f)	1
(e,f)	2

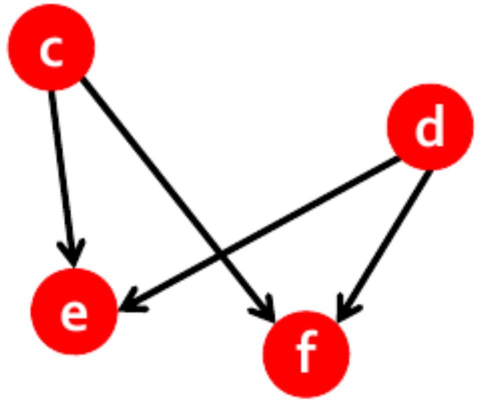
Hence, L(2) is {(b,d),(e,f)}

Third Pass

(b,d,e)	1
(b,d,f)	1
(e,f,b)	1
(e,f,d)	1

Hence, L(3) is { }


a
b
c
d
e


c
d
e
f

PSEUDOCODE

Phase 1

```
let T be the hash table of baskets
count = {}
L(1) = { singleton itemsets }

for (node,t) in T do
     let C(t) be those elements of L(1) which are a subset of basket t
     for candidates c in C(t) do
          count[c] += 1

L(1) = { singleton itemsets whose frequency >= s }
```

Phase 2

```
k = 2

while L(k-1) != 0 do
     count = {}
     graph = {}
     let C(k) be the candidate matrix obtained from L(k-1)xL(k-1)

     for (node,t) in T do
          let C(t) be those elements of C(k) which are a subset of basket t
          for candidates c in C(t) do
               graph[c].append(node) // append the basket name to each itemset
               count[c] += 1

     let L(k) be those elements of C(k) for which count >= s
     k += 1

return L(1)...L(k)
```

MapReduce

Phase 1

This has to be run for the first phase, where k = 1

Mapper

```
Input: <item,t>  // list of all baskets
for node in t do
      emit <node; 1>
```

Reducer

```
Input: <node; 1>
sum = {}

foreach node in Input do
      sum[node]++

foreach (node,freq) in sum do
      if sum >= support then
            emit <node>
```

Phase 2

This has to be run individually for all k-phases, where k >= 2

Mapper

```
Input: L(k-1), T, k
C(k) = L(k-1) x L(k-1) // self join
foreach (node, t) in T do
     let C(t) be those elements of C(k) which are a subset of basket t
     foreach candidate c in C(t) do
          emit <c; node>
```

Reducer

```
Input: <c; node>
sum = {}
graph = {}

foreach (c,node) in Input do
     sum[c] += 1
     graph[c].append(node)

foreach (c,list) in graph do
     if sum[c] >= support then
          emit <c; list>
```

COUNTING TRIANGLES

Why Count Triangles?

If a graph is a social network with n participants and m pairs of “friends,” we would expect the number of triangles to be much greater than the value for a random graph. The reason is that if A and B are friends, and A is also a friend of C, there should be a **much greater chance than average** that B and C are also friends. Thus, counting the number of triangles helps us to measure the extent to which a graph looks like a social network.

We can also look at communities within a social network. It has been demonstrated that the age of a community is related to the **density of triangles**. That is, when a group has just formed, people pull in their like-minded friends, but the number of triangles is relatively small. If A brings in friends B and C, it may well be that B and C do not know each other. As the community matures, B and C may interact because of their membership in the community. Thus, there is a good chance that at sometime the triangle {A, B, C} will be completed.

ALGORITHM

Assuming the graph is represented by its edges, we preprocess the graph as follows -

- Compute the degree of each node. This part requires only that we examine each edge and add 1 to the count of each of its two nodes. The total time required is O(m).

- Create an index on edges, with the pair of nodes at its ends as the key. That is, the index allows us to determine, given two nodes, whether the edge between them exists. A hash table suffices. It can be constructed in O(m) time, and the expected time to answer a query about the existence of an edge is a constant.

- Create another index of edges, this one with key equal to a single node. Given a node v, we can retrieve the nodes adjacent to v in time proportional to the number of those nodes.

CONTD...

Suppose we have a graph of n nodes and m ≥ n edges. For convenience, assume the nodes are integers 1, 2, . . . , n. We call a node a **heavy hitter** if its degree is at least $\sqrt{m}$ A heavy-hitter triangle is a triangle all three of whose nodes are heavy hitters. We use separate algorithms to count the heavy-hitter triangles and all other triangles.

In the following algorithms, we will be using the symbol ^ quite often. We say u ^ v if -

- The degree of u is less than the degree of v, or
- The degrees of u and v are the same, and u < v

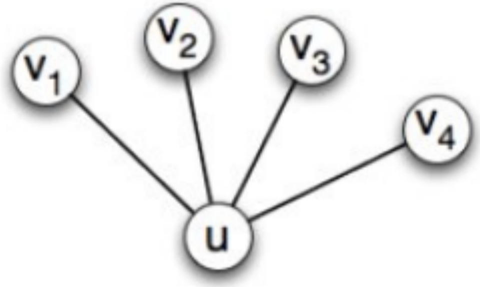
This type of ordering is used to ensure that we count each triangle only one.

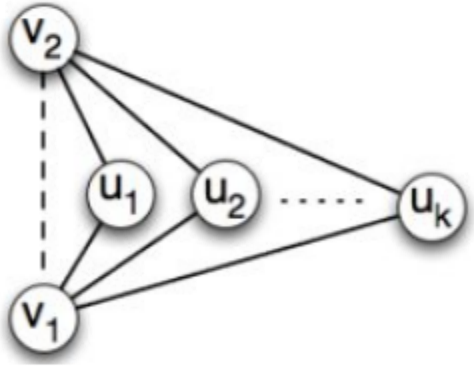
PSEUDOCODE

```
// pre-processing the graph

let G = graph
let degree[0...n] = 0
let edges = {} // a hash with the edge as the key and boolean 1 as the value
let nodes = {} // a hash with the node as the key and a list of adjacent nodes as the value

for edge E in G do          // compute the degree of each node
     degree[E.node1] += 1
     degree[E.node2] += 1


for edge E in G do          // creates an index on edges
     edges[E] = 1


for edge E in G do          // creates a hash for storing adjacent nodes
     nodes[E.node1].append(E.node2)
```

```
// counting heavy hitter triangles

count_heavy = 0
for v1 in 0...n do
     if degree[v1] >= sqrt(m) then
          for v2 in (v1+1)...n do
               if degree[v2] >= sqrt(m) and checkEdge(v1,v2) then
                    for v3 in (v2+1)...n do
                         if degree[v3] >= sqrt(m) and checkEdge(v1,v3)
                              and checkEdge(v2,v3) then
                              count_heavy += 1

// counting other triangles

count_other = 0
for (v1,v2) in edges do
     if degree[v1] >= sqrt(m) and degree[v2] >= sqrt(m) then
          continue
     else if v1 ^ v2 do
          for v3 in nodes[v1] do
               if !checkEdge(v3,v2) do
                    continue

               if v2 ^ v3 do
                    count_other += 1
```

MapReduce





MapReduce

Phase 1

```
Mapper

Input: <(u,v); ∅>
if u ^ v then
     emit <u; v>

Reducer

Input: <u; v>

foreach (u,v) in Input do
     S[u].append(v)

foreach (u,list) in S do
     for (v,w) : v, w ∈ list do
          emit <u; (v,w)>
```

[Ref] Counting Triangles and the Curse of the Last Reducer - Siddharth Suri, Sergei Vassilvitskii - Yahoo! Research

Phase 2

Mapper

```
if Input of type <v; (u,w)> then
      emit <(u,w);v>

if Input of type <(u,v); ∅> then
      emit <(u,v); $>
```

Reducer

```
Input: <(u,w); value>
count = 0
S = {}

foreach (pair, value) in Input do
      S[pair].append(value)

foreach (pair,list) in S do
      if $ ∈ list then
            foreach v in list do
                  if v != $ then
                        count = count + 1

emit <count>
```

CONCLUSION

So, to conclude, we learnt that -

- Communities can be found within graphs by calculating by removing edge based on betweenness scores. Higher the betweenness, lesser chance of it belonging to particular community.

- Bipartite subgraphs give us an idea of the most densely connected communities in any social-network. It's a good way to for finding well-defined subsets directly from the graph.

- Counting triangles gives us an in idea of locality, which can used to judge the connectedness and age of a community.