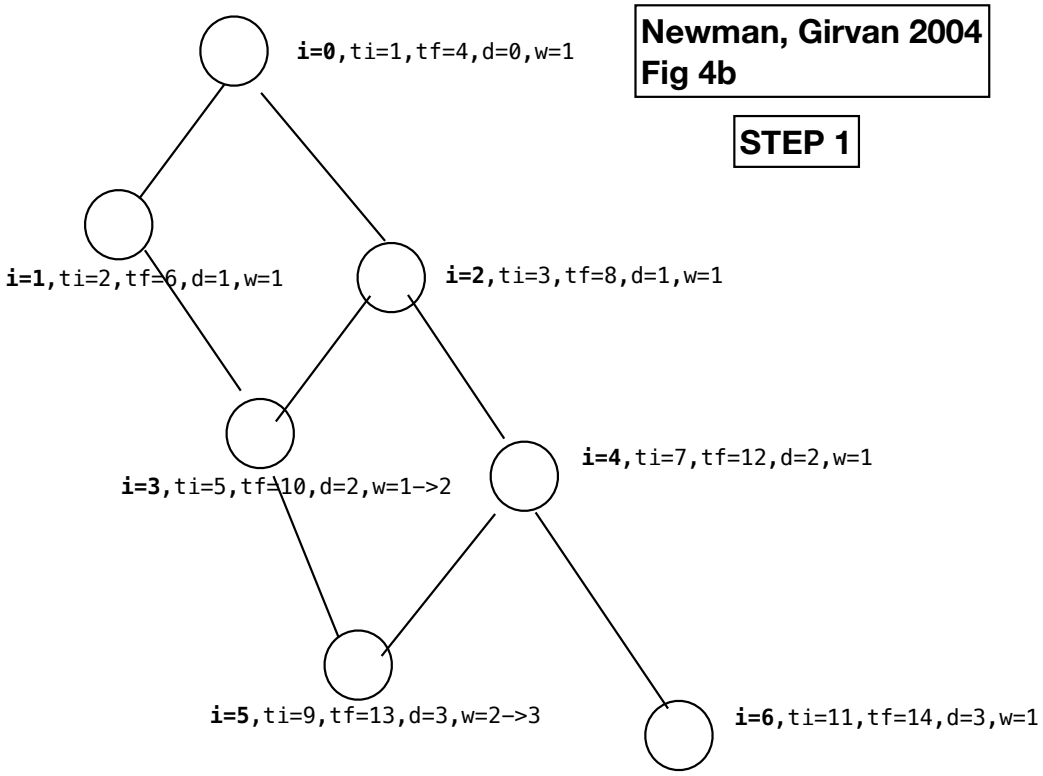
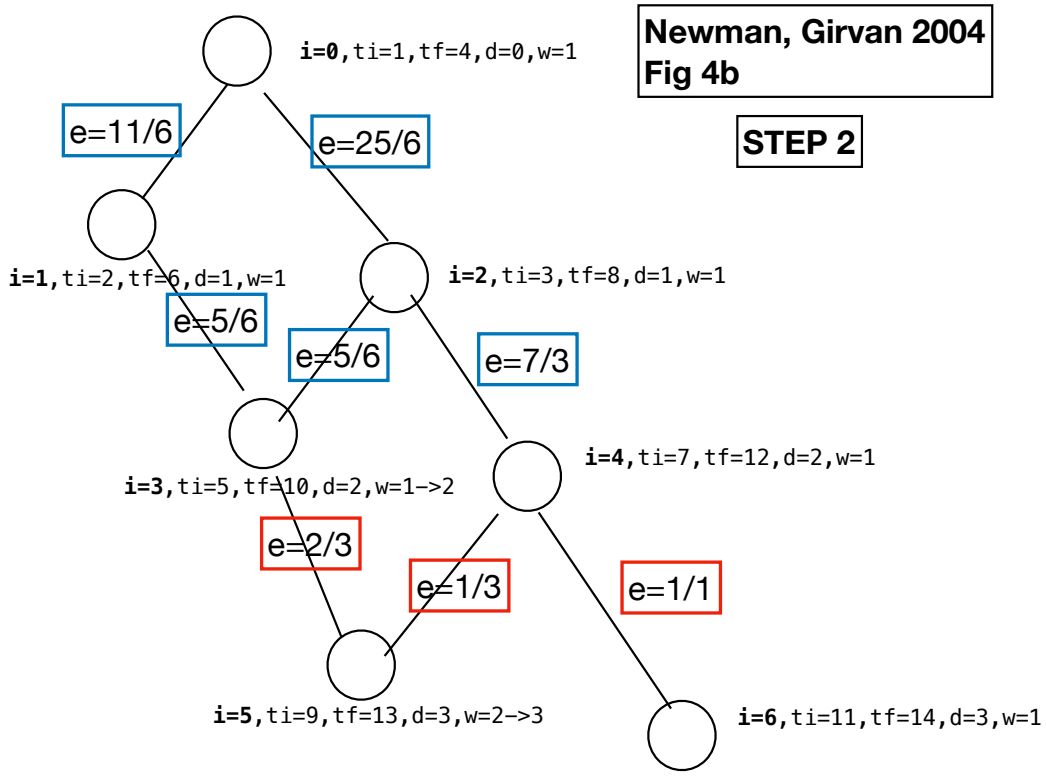


0 (ti=1, tf=4,d=0,w=1) enque: **1** (ti=2,d=1,w=1,p=0), **2** (ti=3,d=1,w=1,p=0)
1 (tf=6) enque: **3** (ti=5,d=2,w=1,p=1)
2 (tf=8) enque: !**3 (w=1+1=>2,p=1,2), 4** (ti=7,d=2,w=1,p=2)
3 (tf=10) enque: **5** (ti=9,d=3,w=2,p=3)
4 (tf=12) enque: !**5 (w=2+1=3,p=3,4), 6** (ti=11,d=3,w=1,p=4)
5 (tf=13, LEAF): –
6 (tf=14, LEAF): –

```
for all vertex {color=0, d=inf, p=-1}
color[s]=1
d[s] = 0; w[s]=1;
p[s] = -1; <=== needs to hold a list now
t=0; ti[s]=++t;
leaf=new Arraylist()
Queue queue = new Queue();
queue.enqueue(s);
while (!queue.isEmpty()) {
    LinkedListNode uNode = queue.dequeue();
    LinkedList neighbors = adjacencyList[uNode.key];
    if (neighbors == null || neighbors.list == null) {
        leaf.add(uNode.key)
        color[uNode.key]=2;
        tf[uNode.key]=++t;
        continue;
    }
    LinkedListNode vNode = neighbors.list;
    while (vNode != null) {
        if (color[vNode.key]==0) {
            color[vNode.key]=1;
            d[vNode.key]=(d[uNode.key]+1;
            w[vNode.key] = w[uNode.key];
            ti[u]=++t;
            queue.enqueue(vNode.key);
        } else if (d[vNode.key]==(d[uNode.key]+1)) {
            w[vNode.key] += w[uNode.key];
        } else {
            assert(d[vNode.key]<((d[uNode.key]+1));
        }
        p[vNode.key] push uNode.key;
        vNode = vNode.next;
    }
    color[uNode.key]=2;
    tf[u]=++t;
}
```



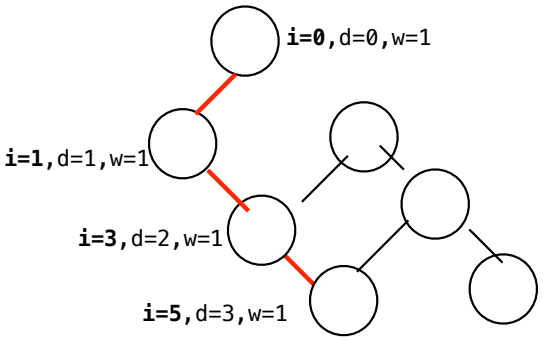
Newman, Girvan 2004
Fig 4b

STEP 2

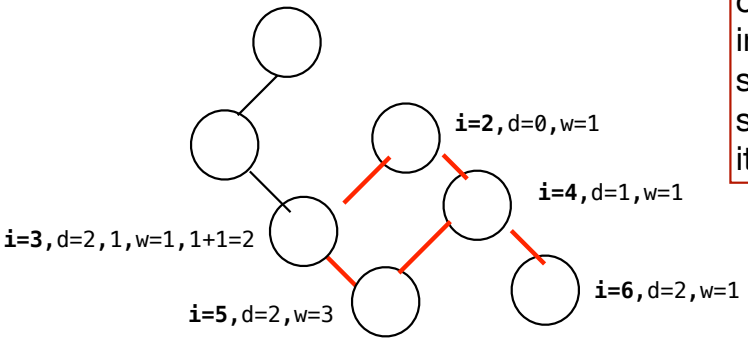
```
queu in progress = 4,3,2,1,2,0, 0
leaf 6: p=4 => e_4_6=1/1
leaf 5: p=3,4 ==> e_4_5=1/3, e_3_5=2/3
     4: p=2 -> e_2_4= (1 + (1/3 + 1)*(1/1)=7/3
     3: p=1,2 -> e_1_3 = (1+(2/3))*(1/2)=5/6
                 e_2_3 = (1+(2/3))*(1/2)=5/6

     2: p=0 -> e_0_2=(1+(5/6 + 7/3))*(1/1)
                    =(1+(19/6))*(1/1)=25/6
     1: p=0 -> e_0_1 = (1+(5/6))*(1/1)=11/6
     0: p = nil
```

```
queue = new Queue();
edges = new TObjectFloatMap<PairInt>();
enqd = new set();
float e;
//NOTE: since only need max edge, will compare below
//      instead of storing in a priority queue/max queue
for (int t : leaf) {
    for (int i : p[t]) {
        if (!enqd.contains(i)) {
            queue.enqueue(i);
            enqd.add(i);
        }
        e = (float)w[i]/(float)w[t];
        edges.put(new PairInt(i,t), e);
    }
}
int i;
float e2;
while (!queue.isEmpty()) {
    i = queue.dequeue();
    LinkedList neighbors = adjacencyList[i];
    assert(neighbors != null);
    assert(neighbors.list != null;
    e = 1;
    LinkedListNode jNode = neighbors.list;
    while (jNode != null) {
        e += (float)w[i]/(float)w[jNode.key];
        jNode = jNode.next;
    }
    for (int ip : p[t]) {
        e2 = (float)w[ip]/(float)w[i];
        e2 *= e;
        edges.put(new PairInt(ip,i), e2);
    }
}
float max; PairInt maxIJ;
iterate over edges to find maxI
```



0 (d=0,w=1) enque: **1** (d=1,w=1,p=0)
1 enque: **3** (d=2,w=1,p=1)
3 enque: **5** (d=3,w=1,p=3)
5 LEAF: –



2 (d=0,w=1) enque: **3** (d=1,w=1,1+1=2,p=1,2), 4(d=1,w=1,p=2)
3 enque: 5 **REPLACES 3->5** (d=1,w=2,p=3)
4 enque: 5 (d=2,w=2,2+1=3,p=3,4), 6 (d=2,w=1,p=4)
5 LEAF
6 LEAF

Newman, Girvan 2004
Fig 4b

iter 2: STEP 1 after 1st max edge cut

Changes needed for algorithm:

1) determine the source nodes which need to iterated over in STEP1 by extracting DFS.predecessors where values == -1.
2) step 1 possibly needs to be edited to change d to a map with key = source vertex and value being a pair of index v and distance. haven't revisited code yet.
3) w has to to hold edge details and delay node sums until needed in order to be able to replace an edge weight for that node's total. instead of w[i] as int array, will need to change to w[i] holding map of sets of {j, value} so can replace an existing key of i, j pair with a subsequent graph source iteration (see edge (3, 5) in lower graph iteration on this page).

Newman, Girvan 2004
Fig 4b

STEP 3

TODO:

(1) need to make wrapper to use Betweenness girvanNewman
for each complete graph calculation (subgraphs handled internal to method).

(2) wrapper then remove max edge

(3) wrapper calculates modularity Q for the girvanNewman Results
(3a) some of the details for modularity are on next page, but missing some of the community member details

need to write the test structure for datasets:
need to implement “randomized fixed number of clusters and edges”

modularity notes,
following http://www.maths.qmul.ac.uk/~latora/report_06.pdf

```
expect a good split is when most edges fall inside the communities,
  and few edges join the communities to each other.
  let E be an nxn symmetric matrix
      where e_i_j is fraction of all edges in the network that link vertices
      in community i to vertices in community j.
  Tr E is the trace of E (= sum of diagonal elements) is
      is the fraction of edges in the network that connect vertices in the
      same community,
  a_i is the row (or column) sums over j,
      and is the fraction of edges that connect to vertices in community i
If the network is such that the probability to have an edge between two sites
   is the same regardless of their eventual belonging to the same
   community, one would have e_i_j = a_i*a_j.
Q is the modularity
   = summation_{over i} of (e_i_i - a_i)^2
   = Tr E - ||E^2||.
     where ||E^2|| is the sum of the elements of E^2.
The modularity measures the degree of correlation between the probability
   of having an edge joining two sites and the fact that the sites belong
   to the same community.
Values approaching Q = 1, which is the maximum, indicate a strong community structure;
   conversely, Q = 0 for a random graph with no community structure.
   Local peaks in the modularity during the progress of the algorithm indicate
   particularly good divisions of the graph.
   The modularity Q corresponding to the groups determined after each split
```

note:

```
  m=|E|
  n=|V|
  S = set of nodes in the cluster
  n_S=|S|
  m_S = number of edges in S = |{(u,v) : u in S, v in S}|
  c_S = number of edges on boundary of S = |{(u,v) : u in S, v not in S}|
  d(u) is the degree of node u

modularity: measures internal (and not external) connectivity, but it does so
   with reference to a randomized null model.
   random graphs have high-modularity subsets and
   there is a size scale below which modularity cannot identify communities
   Modularity: (1/(4*m))*(m_S - E(m_S))
      where E(m_S) is the expected number of edges between the nodes in set S
      in a random graph with the same node degree sequence.
Modularity ratio: m_S/E(m_S) is alternative definition of the modularity,
   where we take the ratio of the number of edges between the nodes of S
   and the expected number of such edges under the null-model.
Volume: summation_{all u in S} of d(u) is sum of degrees of nodes in S
Edges cut: c_S is number of edges needed to be removed to disconnect nodes in S
   from the rest of the network

A general observation is that modularity tends to increase roughly monotonically towards
   the bisection of the network.
On the other hand, the modularity ratio tends to decrease towards the bisection of the network.
   Results in Figure 6 demonstrate that, with respect to the modularity,
   the “best” community in any of these networks has about half of
   all nodes; while,
   with respect to the modularity ratio, the “best” community in any of these networks
```