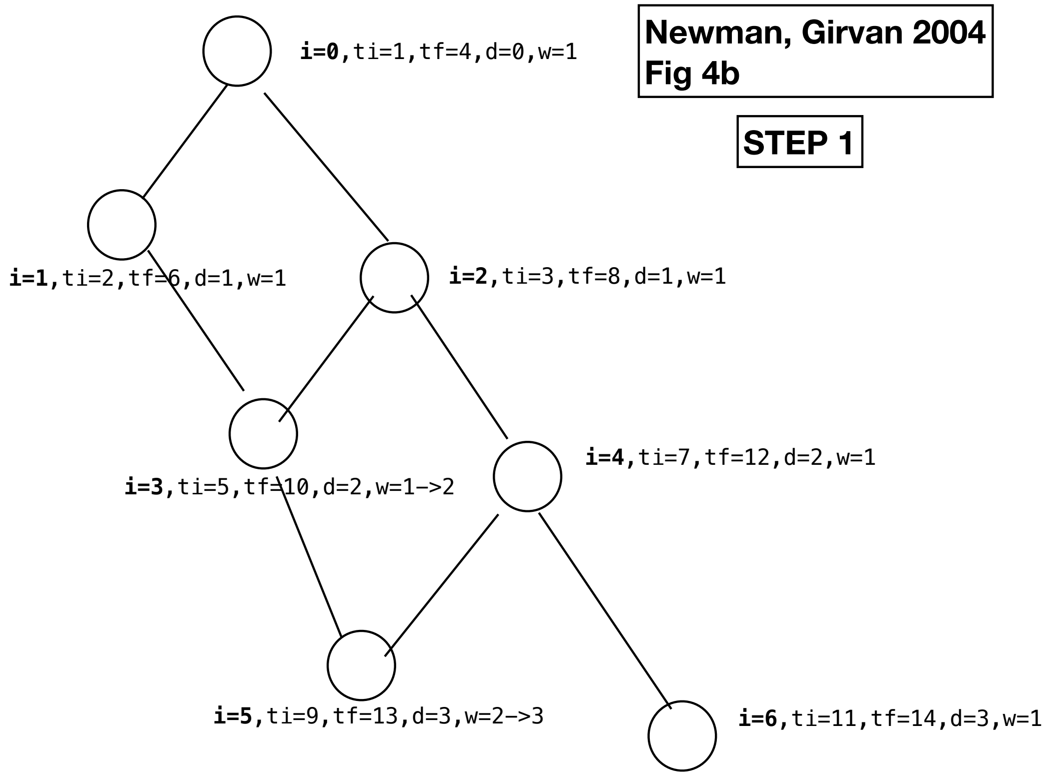


Newman, Girvan 2004
Fig 4b

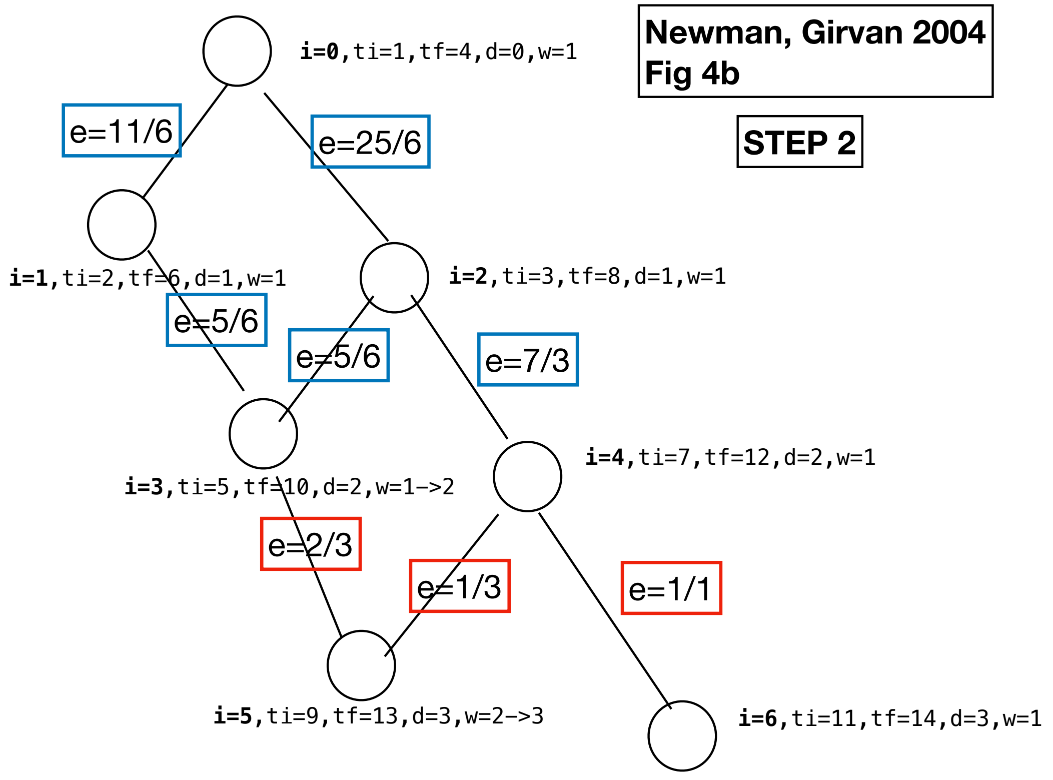
STEP 1

```
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



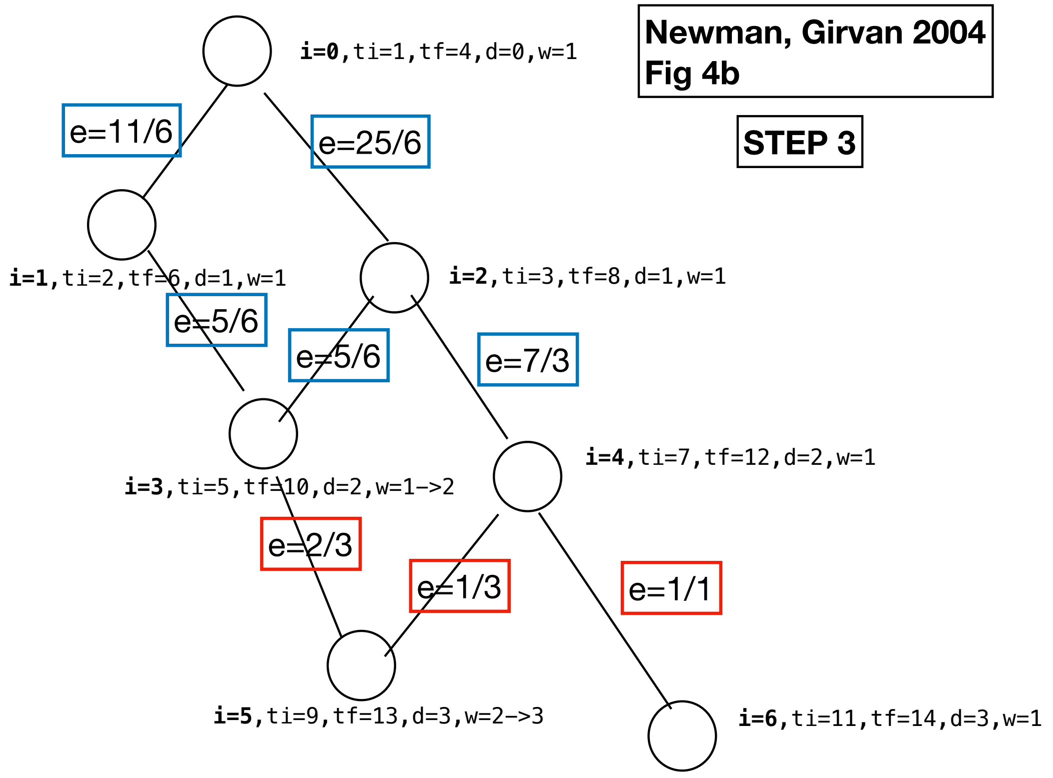
```
queu in progress = 4,3,2,1,2,0, 0
leaf 6: p=4 => e_4_6=1/1
leaf 5: p=3,4 ==> e_4_5=1/3, e_3_5=2/3
     4: p=2 -> e_2_4= (1 + (1/3 + 1)*(1/1)=7/3
     3: p=1,2 -> e_1_3 = (1+(2/3))*(1/2)=5/6
                 e_2_3 = (1+(2/3))*(1/2)=5/6

     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
```

Newman, Girvan 2004
Fig 4b

STEP 3



i=0,ti=1,tf=4,d=0,w=1
e=11/6
e=25/6
i=1,ti=2,tf=6,d=1,w=1
i=2,ti=3,tf=8,d=1,w=1
e=5/6
e=5/6
e=7/3
i=3,ti=5,tf=10,d=2,w=1->2
i=4,ti=7,tf=12,d=2,w=1
e=2/3
e=1/3
e=1/1
i=5,ti=9,tf=13,d=3,w=2->3
i=6,ti=11,tf=14,d=3,w=1

```
TODO:

need to remove max edge

need to wrap step 1 and 2 in an iteration and handle the
    splitting of graph into subgraphs

need to implement the modularity Q calculation for subdivision of
    communities

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

```
pg 181 from http://www.maths.qmul.ac.uk/~latora/report_06.pdf
To know which of the divisions is the best one for a given network,
   i.e. where to cut the hierarchical tree, one can use the
   modularity Q, a quantity introduced in Ref. [51] and defined in
   27 the following way. Let us suppose that we want to test the goodness of a
   subdivision of G in n given communities.

   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
```