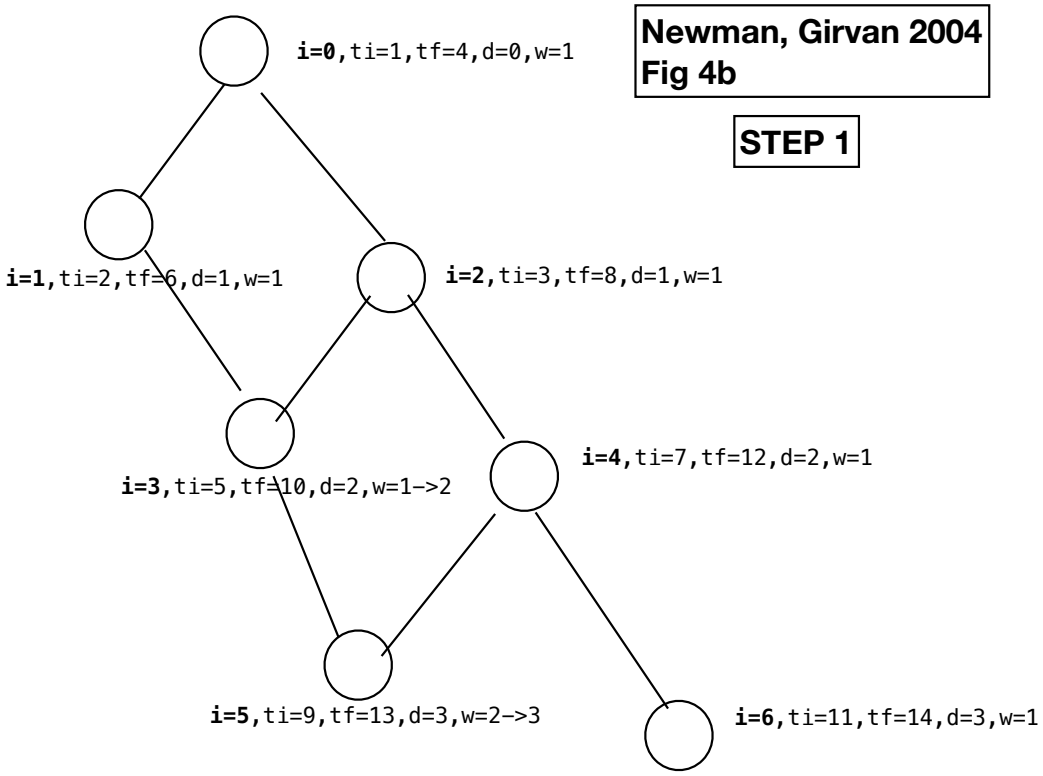
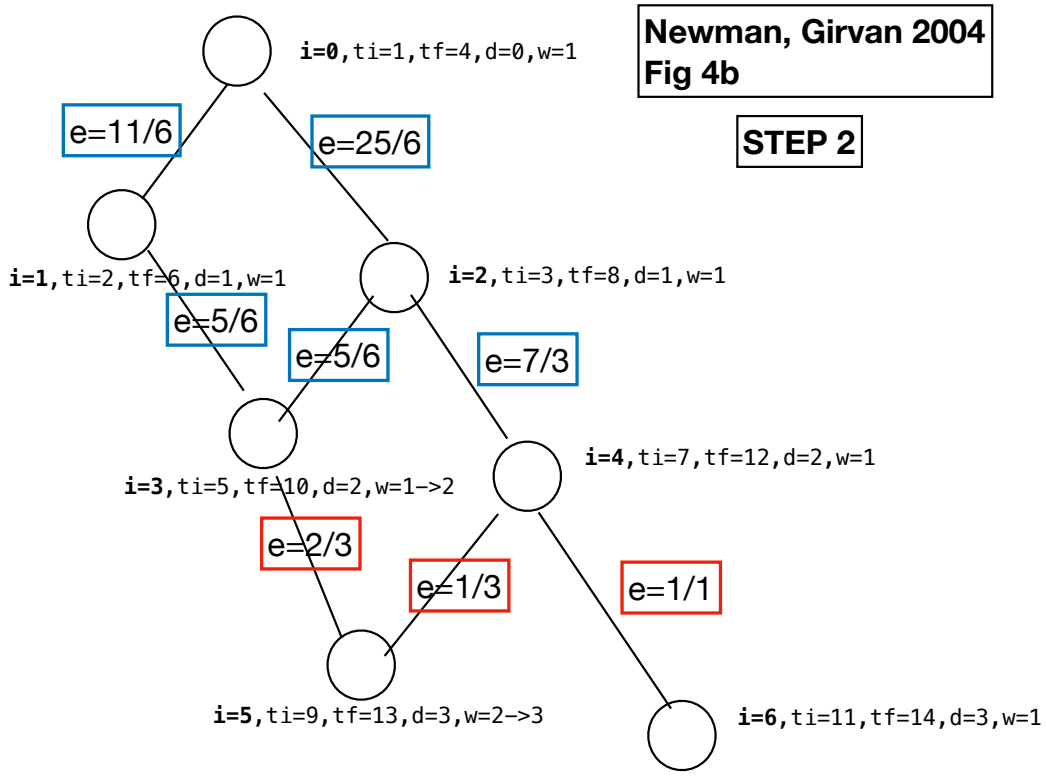


0 (ti=1, tf=4,d=0,w=1) enque: **1** (ti=2,d=1,w=1,p=0), **2** (ti=3,d=1,w=1,p=0)
1 (tf=6) enque: **3** (ti=5,d=2,w=1,p=1)
2 (tf=8) enque: !**3 (w=1+1=>2,p=1,2), 4** (ti=7,d=2,w=1,p=2)
3 (tf=10) enque: **5** (ti=9,d=3,w=2,p=3)
4 (tf=12) enque: !**5 (w=2+1=3,p=3,4), 6** (ti=11,d=3,w=1,p=4)
5 (tf=13, LEAF):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

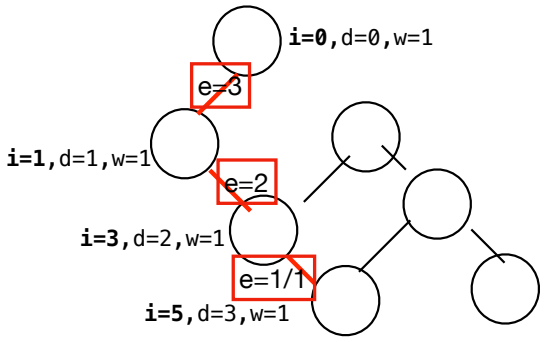
```
for all vertex {color=0, d=inf, p=-1}
color[s]=1
d[s] = 0; w[s]=1;
p[s] = -1; <=== needs to hold a list now
t=0; ti[s]=++t;
leaf=new Arraylist()
Queue queue = new Queue();
queue.enqueue(s);
while (!queue.isEmpty()) {
    LinkedListNode uNode = queue.dequeue();
    LinkedList neighbors = adjacencyList[uNode.key];
    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;
}
```



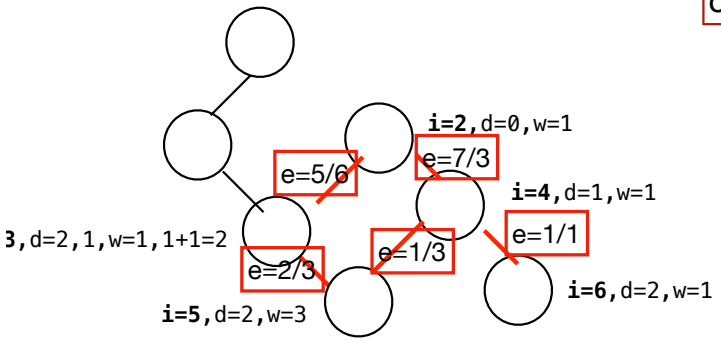

```
queu in progress = 4,3,2,1,2,0, 0
leaf 6: p=4 => e_4_6=1/1
leaf 5: p=3,4 ==> e_4_5=1/3, e_3_5=2/3
     4: p=2 -> e_2_4= (1 + (1/3 + 1)*(1/1)=7/3
     3: p=1,2 -> e_1_3 = (1+(2/3))*(1/2)=5/6
                 e_2_3 = (1+(2/3))*(1/2)=5/6

     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
2:4=(1+(1)+(1/3))*(1/1)=7/3
2:3=(1+(2/3))*(1/2)=5/6
```

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 my version of DFS.predecessors where values == -1.
2) step 1 needs to be iterate over each root node.

also needs a weight array outside of the root iteration to hold all path counts for all root traversals.

also needs to be edited so that each root STEP2 edge weight calculation includes the previous path counts