Task 4

Using a heap-like structure

Using a heap-like structure `hp3` to represent unmarked nodes based on `m` values is good, because in each iteration we need to remove a node from the unmarked list, and find the m-least unmarked node from the remaining. Right now the removal takes a time complexity of O(1), but searching takes O(N), so in total they takes O(N). But with a min-heap, removal takes O(log(n)) and searching takes O(1), in total O(log(n)). The algorithm will be much more efficient with this refinement.

Other variables to be added

No other variables need to be added other than `hp3`, which is a list of (m value, node id) pairs, maintained as a min-heap using python built-in `heapq` functions.

Statements affected

1. On initialisation of m: `m= [INFTY]*N; m[0]= 0`

The heap needs to be initialized, with the pairs (m[i], i) for all unmarked nodes: `hp3 = [(m[i], i) for i in range(N)]; heapify(hp3)`

2. On updating of m: `m[nTo]= newD`

The heap also needs to be updated with the new m value: `heappush(hp3, (newD, nTo))`. It does not matter if in the heap there already exists a pair for the same node, because its priority-in-heap cannot be higher than newD. By the time the old pair is poped, `nTo` will have been marked (immediately after the higher priority pair for the same node was poped), so this pair will be ignored.

3. Getting the m-least unmarked node: statements `for n in range(N): if un1[n]: if m[n] <=minD: tn= n; minD= m[n]`

Now we can replace these statements by reading the m-least unmarked node directly from the heap. Note that the heap may contain marked nodes, so we need to keep popping until we get an unmarked one.

Code

```
import sys; INFTY= 1000
from heapq import heapify, heappush, heappop
print "How many nodes? ",
N= int(sys.stdin.readline()) # N= number of nodes.
print "Nodes are 0..%d, with initial node %d." % (N-1, 0)

# Three white-space separated integers representing (from,dist,to) per line.
print "Now enter the edges:"
G2 = [[] for i in range(N)]
for line in sys.stdin.readlines():
    nFrom, dist, nTo = map(int, line.split())
    G2[nFrom].append((dist, nTo))
un1 = [1]*N; nun1 = N   # un1[i] == 1 for all nodes: all nodes are
unmarked
m= [INFTY]*N; m[0]= 0   # m:=  initially INFTY except for initial node
#>>
hp3 = [(m[i], i) for i in range(N)]  # [(m[0], 0)]
heapify(hp3)
#<<
tn= 0                   # tn:= initial node

# Initialisation has established the invariants:
#   I1--- For all marked nodes, m has the least distance from 0. // A.
#   I2--- For all unmarked nodes, m has least all-marked distance from 0.
// B.
#   I3--- Node tn is unmarked, and is m-least among the unmarked nodes. //
C.

while 1:
    un1[tn] = 0; nun1 -= 1
    if nun1 == 0: break

    # Re-establish I2. // G.
    for (dist, nTo) in G2[tn]:
        if un1[nTo]:
            newD= m[tn]+dist
            if newD<m[nTo]:
                m[nTo]= newD
#>>
                heappush(hp3, (newD, nTo))
#<<

    # Re-establish I3. // H.
    minD= INFTY
#>> for n in range(N):
#>>   if un1[n]:
#>>     if m[n]<=minD: tn= n; minD= m[n]
    while hp3:
        dist, n = heappop(hp3)
        if un1[n]:
            minD, tn = dist, n
            break
#<<

    if minD==INFTY: break # All remaining nodes unreachable. // I.
###
#   I1,I2 and m is INFTY for all nodes in un1. // J,K,L,M.

print "Least distances from Node 0 are:"
for n in range(N):
    if m[n]!=INFTY: print "Distance to Node %d is %d." % (n,m[n])
    else:           print "Node %d is unreachable." % n
```