Brandes algorithm

These notes supplement the notes and slides for Task 11.
They do not add any new material, but may be helpful in understanding the Brandes algorithm for calculating node betweenness centrality. See Brandes' papers for further details (URLs are in the task instructions). As in the lectures, I use the notation from Brandes (2008) but the original paper is Brandes (2001).

Node betweenness centrality: the definition.

Betweenness centrality for a node v is defined in terms of the proportion of shortest paths that go through v. Specifically:

1. Assume a directed, unweighted, connected graph $G =< V, E >$.
2. Define $\sigma(s, t)$ as the number of shortest paths between nodes s and t.
3. Define $\sigma(s, t|v)$ as the number of shortest paths between nodes s and t that pass through v.
4. $C_B(v)$, the betweenness centrality of v is defined as:

$$C_B(v) = \sum_{s,t \in V} \frac{\sigma(s, t|v)}{\sigma(s, t)}$$

   If $s = t$, then $\sigma(s, t) = 1$. If $v \in s, t$, then $\sigma(s, t|v) = 0$.

Brandes' algorithm is for the case where we want to calculate this efficiently for every node.

Naive approach

The approach taken here to explaining Brandes' algorithm, is to start by discussing a very naive implementation and progressively refine it. Taking the definition of $C_B(v)$ above, a naive approach is as follows:

1. For every node v in V, set $C_B(v) = 0$.
2. For each node s in V, use a BFS algorithm to find all the shortest paths between s and all other nodes. Store all these paths for each pair s,t.
3. For each pair s,t, count the number of times v appears in the stored paths to give $\sigma(s, t|v)$ and divide by the total number of paths between s and t (i.e., $\sigma(s, t)$). Add the result to $C_B(v)$.
4. $C_B(v)$ gives the final result.

An BFS algorithm to find all shortest paths with unweighted graphs is shown below, using the same notation as in the Brandes (2008) pseudocode. In later refinements of the Brandes algorithm, this will be integrated into the main code.

1. Initialization: for each node w:
    1. Mark w as unvisited by setting dist[w] (the distance between s and the node w) to infinity.
    2. Set Pred[w] (nodes that immediately precede w on a shortest path from s) to the empty list.
    3. Set Paths[w] (the list of all shortest paths from s to w) to the empty list.

    Starting node:
    1. Choose the starting node s and put it on the queue Q.
    2. Set dist[s] to 0.
2. while Q is not empty, do:
    1. dequeue v from Q
    2. For each node w such that there is an edge in E from v to w, do
        1. if dist[w] is infinity, then
            1. set dist[w] to dist[v] + 1
            2. enqueue w
        2. if dist[w] = dist[v]+1 then
            1. append v to Pred[w]

3. Collect all paths by following Pred[t] back to s for each t, storing paths on Paths[w]. This step won't be used in the more refined versions of the algorithm, so is not elaborated here.

Note that, since we are doing a BFS starting at s, we never need to reset dist[w].

You may find it useful to think of each interation with a different s in terms of what would happen if the graph were a physical net, with all links of equal length, which you picked up by each successive s. The s node is at the top, and some nodes are hanging at the bottom, with no nodes below them on shortest paths. Informally, I will refer to these as terminal nodes. The backward phase, where we collect paths, starts once we've reached all the terminal nodes.

Improving on the naive approach

There are a number of ways we might intuitively think of improving on the naive approach. The steps below are chosen to move towards the Brandes algorithm.

Storage efficiency

The naive approach is very expensive in terms of storage. However we don't need to save anything about the paths between s and t once we've updated $C_B(v)$

for each vertex v on those paths. Hence we could refine our naive algorithm as follows:

1. For every node v in V, set $C_B(v) = 0$.
2. For each node s in V:
    1. Use a BFS algorithm to find all the shortest paths between s and all other nodes. Store the paths for each target t.
    2. For each t, for each vertex w that occurs on one of the stored paths, count the number of times w appears in total to give $\sigma(s,t|w)$ and divide by the total number of paths between s and t (i.e., $\sigma(s,t)$). Add the result to $C_B(w)$.
3. $C_B(w)$ gives the final result.

Integration with the shortest paths algorithm

We observe that the BFS algorithm involves spreading out from s to find the shortest paths up to the terminal nodes (i.e., the ones which don't have any following nodes on the shortest paths from s) and then stepping back via the saved predecessor nodes to actually output the paths for the terminal nodes and all the previous nodes. Therefore, we are actually going back to s from each node t through all the nodes v which are on the shortest path between s and t. We could therefore add up the shortest paths at that point, rather than saving them all and then checking whether v is a member.

For instance, we could use a 2-dimensional array to store values for $\sigma(s,t|v)$ (two dimensional because s is constant for our use of the array) and each time we reach a node v on the return path from a node t we increment the array.

1. For every node v in V, set $C_B(v) = 0$.
2. For each node s in V:
    1. set $S(v,t)$ to zero for all nodes v and t in V.
    2. Use the BFS algorithm, as above, to reach each target t from s.
    3. In the backward phase, increment $S(v,t)$ as appropriate when each node v is reached, rather than creating full paths.
    4. At the end, divide each $S(v,t)$ by the total number of paths between s and t (i.e., $S(s,t)$). Add the result to $C_B(v)$.
3. $C_B(w)$ gives the final result.

Recursive calculation

The main difference between what we have above and the Brandes algorithm is that the latter makes use of a recursive step in the backward phase to allow direct calculation of the ratios for each v on the basis of its successor nodes on the shortest paths to every following t.

For now, let's simply pretend we have such a function, which we call Magic, and a value $\delta(v)$ such that the following conditions hold:

1. $\delta(t) = 0$ if t is a terminal node (as described above).
2. We can increment $\delta(v)$ via Magic every time we reach v from a node w on the backward phase (i.e., v immediately precedes w in a shortest path from s) based on the values of $\delta(w)$.
3. After we have finished with all the w values, $\delta(v)$ can be straightforwardly accumulated into $C_B(v)$.

Here is revised pseudocode under this assumption:

1. For every node v in V, set $C_B(w) = 0$.
2. For each node s in V:
    1. set $\delta(v)$ to zero for all nodes v in V.
    2. Use the BFS algorithm (much as before, differences in bold below) while Q is not empty, do:
        1. dequeue v from Q **and push v onto a stack S**
        2. For each node w such that there is an edge in E from v to w, do
            1. if dist[w] is infinity, then
               set dist[w] to dist[v] + 1
               enqueue w
            2. if dist[w] = dist[v]+1 then
               set $\sigma(s, w)$ **to** $\sigma(s, w) + \sigma(s, v)$
               append v to Pred[w]
    3. while S is not empty, pop w off S
        1. for all nodes v in Pred(w) set $\delta(v)$ to $\delta(v) + \text{MAGIC}(\delta(w))$.
        2. unless $w = s$, set $C_B(w) = C_B(w) + \delta(w)$.
3. $C_B(v)$ gives the final result.

It will turn out that MAGIC requires that we know the number of shortest paths between s and each node v, so we create these values in the forward phase of the BFS as shown (i.e., $\sigma(s, w)$). We also need to make sure that the nodes are visited in the correct order on the backward step, which we do by putting the dequeued elements of Q onto a stack in the forward pass, and then visiting the nodes in the order they are popped off the stack in the backward pass.

Brandes' algorithm

We are now at the point where we essentially just need to describe MAGIC and δ to have a complete account of the Brandes pseudocode. First we define δ, then we look at a special case, where MAGIC is simple, and finally we look at the full version of MAGIC.

Dependencies

In Brandes' algorithm, the ratio of the shortest paths between s and t that go through v and the total number of shortest paths between s and t is called the 'pair-wise dependency':

$$\delta(s,t|v) = \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

So:

$$C_B(v) = \sum_{s,t\in V} \delta(s,t|v)$$

The one-sided dependency is defined as:

$$\delta(s|v) = \sum_{t\in V} \delta(s,t|v)$$

and therefore:

$$C_B(v) = \sum_{s\in V} \delta(s|v)$$

The point of doing this is as outlined above: $\delta(s|v)$ can be computed recursively, on the basis of the values $\delta(s|w)$ for the nodes which follow v on shortest paths from s (i.e., without iterating through all t for each v). Since we are always calculating $\delta(s|v)$ for a particular s, we can just write $\delta(v)$.

Tree MAGIC

If the vertices and edges of all shortest paths from s form a tree, then it will hopefully be intuitively clear that $\delta(v)$ can be computed simply. (This is illustrated in Figure 1 on p7 in:

Brandes (2001) http://www.algo.uni-konstanz.de/publications/b-fabc-01.pdf

Look at this figure if you get confused by the following!)

In a little more detail:

1. Assume there is exactly one shortest path from s to each node t (this is equivalent to the tree condition).
2. For any vertex v and any target t, either v lies on the (unique) shortest path between s and t, in which case $\delta(s,t|v) = 1$, or does not lie on the path, in which case $\delta(s,t|v) = 0$.
3. For any vertex w, such that v immediately precedes w on shortest paths from s, v will lie on the shortest path from s to w. And, for any node x, such that w immediately precedes x on shortest paths from s, v will lie on the shortest path from s to x. And so on.
4. Hence we can simply total all the one-sided dependencies relating to paths that go to nodes beyond each node w, and add one for every w, giving:

$$\delta(v) = \sum_{w} (1 + \delta(w))$$

Ultimate MAGIC

The situation with the non-tree case, where there are alternative shortest paths that bypass v, is more complex. Thinking about this in terms of the ratios which are being accumulated, the issue is that some proportion of the shortest paths to nodes beyond v will go through v but others will not, and we need a way to determine this ratio. This is illustrated in Figure 2 in:

Brandes (2001) http://www.algo.uni-konstanz.de/publications/b-fabc-01.pdf

I recommend that you look at this figure.

Brandes' equation for MAGIC is:

$$\delta(v) = \sum_{w} \frac{\sigma_{s,v}}{\sigma_{s,w}} (1 + \delta(w))$$

There are separate situations to consider, depending on whether the bypassing edge arrives at one of the nodes w or beyond it. The intuition is that it is only the situation where the bypassing edge arrives at one of the w that we have to worry about. In other cases, the δs of the ws already incorporate the ratios of the shortest paths appropriately. Thus we only need to look at the ratio of the paths going between s and v compared with those going between s and w to capture the extent to which there are bypassing paths going to w. The proof that this is correct is given in Brandes (2001) but is a little messy (because he has to incorporate the bypassing edges), so will not be described in any more detail here.

Hence, in the pseudocode above, we replace the lines:

1. for all nodes v in Pred(w) set $\delta(v)$ to $\delta(v) + \text{MAGIC}(\delta(w))$.

with

1. for all nodes v in Pred(w) set $\delta(v)$ to $\delta(v) + \frac{\sigma_{s,v}}{\sigma_{s,w}}(1 + \delta(w))$.

the MAGIC disappears and we have Brandes' algorithm.