# Lecture 5: Centrality measures*

Eduardo López

The number of triangles, v-shapes, and degree associated with a node capture the local importance of a node. Clearly, this is essential information we require to understand a network. On the other hand, there are other dimensions of the importance of a node (or link) that cannot be captured by purely local measures. In this lecture, we focus on an entire family of importance measures called centralities. There are many flavors of centrality but we are not in a position to study them extensively. Thus, we pick a few of the emblematic ones that can serve as a springboard to understand others.

## 1 Bare betweenness centrality

Consider the role of a node as an intermediary for communication. In such a role, an individual node is part of the paths between other nodes. If a node that acts as an intermediary in a large number of paths is affected in some way, communication in the entire network would likely see effects.

But how could one get a true sense of the possible impact involved in the loss of a key node to a network? Is degree enough? Not necessarily. Degree works in many cases as a measure of importance not because it truly measures the functions performed by a node, but because degree and some of these node functions are sometimes correlated.

These thoughts indicate that what one needs to do is to consider a direct measurement that focuses on the intermediation performed by a node. A generic treatment of this question requires thinking about it in a similar way to a data network such as the Internet. But this generic approach has one natural limit that is generally taken to address social network analysis.

Let us think of what happens when one node (imagine a computer) tries to send information to another: information packets departs from the origin

*adapted from my undergraduate networks book

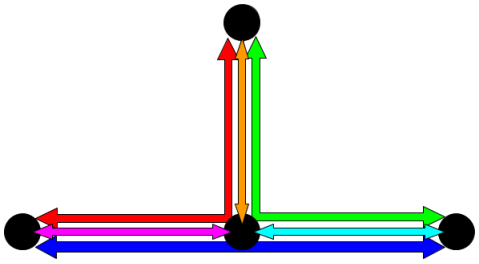

Figure 1: Illustration of a static routing protocol. Each color represents a fixed path information must follow when going between a particular pair of origin and destination nodes.

node and, according to some rules that attempt to move the packets towards their destination, these packets are passed from node to node on the network until they either reach their destination or die along the way as they get lost or "time-out". On the Internet many nodes are sending information, and also many are receiving it. Some nodes act mostly as intermediaries sending the information through but not necessarily being the creators or receivers of the information. Nodes that act as intermediaries for many, many others play a particularly critical role.

To measure the intermediation role, we adopt a counting scheme: the number of paths that visit a node measures the intermediary role of a node. This general concept is called *betweenness centrality.* There is a very concrete definition for it that we will present in a moment. But first, let us introduce the relevant technical elements.

A *static routing protocol* is a list of *fixed* paths $\{\wp_{o,d}\}_{o,d\in G}$ along a network $G$ that specifies how data from an origin node $o$ is supposed to travel to a destination node $d$. A connected network of $n$ nodes should specify $\binom{n}{2}$ paths since there are the same numbers of pairs $o, d$ of origins and destinations in the network; note that this results applies to undirected and directed networks. Thus, the routing table indicates how all information is meant to flow in the network. In a network with a static routing protocol, each node needs to know only what is the next node it has to send information to when it determines the destination of a visiting packet, regardless of the origin or the destination (unless the node in question *is* the destination) of the information. Figure 1 illustrates these concepts.

Now, any given node $i$ in the network may or may not be visited by any of the paths in $\{\wp_{o,d}\}_{o,d\in G}$. Let us define the quantity $b_i$ as the number of

paths of the routing protocol that visit $i$. It is traditional to say that when the paths in the routing protocol are *specifically the shortest paths*, $b_i$ is the *betweenness centrality of* $i$. In social network analysis, this is the norm, and we have

**Definition 1.** *Given the set of shortest paths between all pairs of origin* $o$ *and destination* $d$ *nodes in a network* $G$*, the* **bare betweenness centrality** *(or simply betweenness)* $b_i$ *of node* $i$ *is the number of said shortest paths that visit node* $i$*.*

Let us now try to understand the meaning of the value $b_i$ for node $i$. Consider that node $i$ was to fail or be removed from the network without having a replacement process in place. One could therefore ask what is affected by the removal of $i$. Since $b_i$ counts the number of shortest paths passing through $i$, the removal of $i$ means that $b_i$ paths in the network are eliminated.[1]

The calculation of $\{b_i\}_{i \in G}$ for all nodes of $G$ is involved and requires computational methods, but can be performed in a reasonable amount of time. Formally speaking, we can write the following expressions. If node $i \in \wp_{o,d}$, then

$$b_i = |\{\wp_{o,d} | i \in \wp_{o,d} \forall o, d \in G\}| \tag{1}$$

which basically says: "$b_i$ is equal to the number of elements (size) of the set of paths that visit $i$ out of all possible paths between all origins and destinations in the network $G$." This is a formal definition but does not give us an equation, so we do not use it for calculation except in small networks, as we shall see in Exmps. 1 and 2.

Betweenness itself is a family of definition. Our definition of "bare" betweenness is in fact different (although very similar in what it measures) to other definitions, but it is the simplest we can define. But at this point, as a reference, we state the following:

**Remark 1.** *In the literature, the term* ***bare betweenness*** *does not exits. This is because our version of betweenness is different to the rest of the definitions in use. They are all very similar to our definition, though. The main difference is that we include in the betweenness of a node the cases when it is an origin/destination of a path. It makes our definition simpler to state. However, be aware of this distinction. We will also call bare betweenness*

[1] In reality, this does not necessarily mean that now there are $b_i$ pairs of nodes that cannot communicate as data routing has contingency methods that would update the routing tables. However, it does mean that the new paths introduced to substitute for the loss of $b_i$ paths are likely to be longer and hence less efficient.

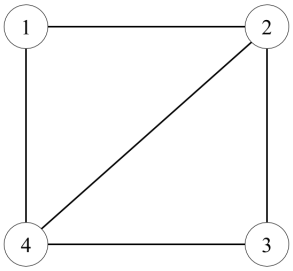


Figure 2: Exmp. 1

*simply betweenness in a number of occasions, but always keep in mind any possible differences in definitions. Finally, just for reference, bare betweenness is very similar to a betwenness definition called* ***load betweenness****.*

Let us now apply the idea behind the definition associated with Eq. 1 in the following example.

**Example 1.** *Determine the betweenness of the nodes in Fig. 2.*

*To tackle this problem, the most logical straightforward approach is to identify all pairs of origin and destination nodes (how many such pairs are there?), the shortest path between them, and then count the nodes visited along the way, including origin and destination. Note that we have to make a choice about what to do when there are more than one shortest path of equal length between origin and destination. Let us write down the pairs:*

$$1,2 \quad 1,3 \quad 1,4 \quad 2,3 \quad 2,4 \quad 3,4.$$

*Now, for each we can identify their shortest paths. Let us start by indicating* ***all*** *shortest paths for each pair and then deciding what to do about the redundancies. Thus,* $1,2$ *have a single path directly linking them, of length* $\ell_{1,2} = 1$*.*

| *pair* | *path(s)* | *length* |
|---|---|---|
| $1,2$ | $1 \to 2$ | $\ell_{1,2} = 1$ |
| $1,3$ | $1 \to 2 \to 3; 1 \to 4 \to 3$ | $\ell_{1,3} = 2$ |
| $1,4$ | $1 \to 4$ | $\ell_{1,4} = 1$ |
| $2,3$ | $2 \to 3$ | $\ell_{2,3} = 1$ |
| $2,4$ | $2 \to 4$ | $\ell_{2,4} = 1$ |
| $3,4$ | $3 \to 4$ | $\ell_{3,4} = 1$ |

*Note that pair* $1, 3$ *have two shortest paths (of length 2) connecting them. One of these paths goes through node 2, and then another one goes through node 4. These nodes are passing-through nodes (or nodes along the way) which contributes to their betweenness.*

*Now, there are two possible broad strategies here: either take into account the two paths when counting betweenness, or pick one of the two paths. The choice may affect the betweenness values of all other nodes. Here are the two (three) possible answers, taking into account that* $b_i$ *counts the number of paths that visit node* $i$*:*

***Taking all paths into account:***

$$b_1 = 4 \quad b_2 = 4 \quad b_3 = 4 \quad b_4 = 4.$$

***Taking only one path between 1,3:***

$$\begin{array}{l} \textit{Only path: } 1 \to 4 \to 3: \\ \quad b_1 = 3 \quad b_2 = 3 \quad b_3 = 3 \quad b_4 = 4. \\ \textit{Only path: } 1 \to 4 \to 3: \\ \quad b_1 = 3 \quad b_2 = 4 \quad b_3 = 3 \quad b_4 = 3. \end{array}$$

*The choice of path affects the betweenness. Clearly chosing to go through node 2 adds one unit of betweenness to 2; making the other choice adds a unit to node 4. Also, using* ***both*** *paths adds an extra unit to the remaining nodes (!) as the extra path visits various nodes along the way.*

This example is typical of the usual situation. The most important statement to make at this point is this

**Remark 2.** *When using any betweenness algorithm,* ***make absolutely certain you know how it counts paths****. Try the algorithm with a network structure you know, and that possibly has pairs of nodes with multiple shortest paths. This will allow you to learn its behavior.*

Now, as indicated when we introduced Eq. 1, the application of this definition as we have done in the previous example is not usually accomplished by hand, and algorithms are involved. This is covered in the computational tools section of this chapter. There are, however, situations when betweenness can be formally calculated, and these help us develop intution about bare betweenness in more detail, and also allow us to check algorithms we would put to use. Here's an example

**Example 2.** *One of the few examples in which we can calculate betweenness centrality formally is on a line network. To be general, assume it is a line*

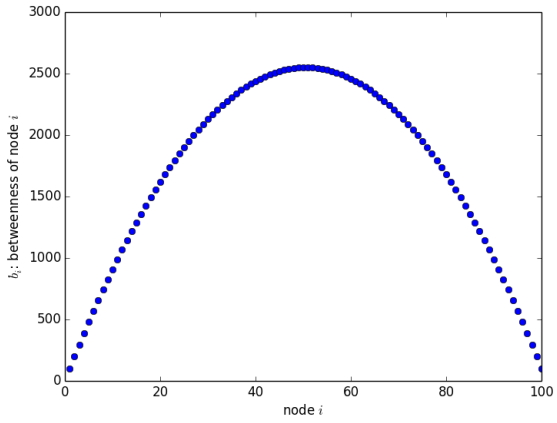


Figure 3: Value for the betweenness $b_i$ of node $i$ in a line of $n = 100$ nodes. Note that the plot is symmetric around the midpoint, as would be expected (the line is the same to the left or the right from the midpoint).

*of $n$ nodes, and let us label the nodes $\{1, 2, \ldots, n\}$. In the line, there are no ambiguities about how to go from one node to another.*

*Consider a node $i$ somewhere in the bulk of the line. Let us count the paths that involve $i$. First, there are paths that go between $i$ and the other $n-1$ nodes in the network. Then, there are the paths that go through $i$ without stopping there. To the left of $i$ one finds the nodes $1, 2, \ldots, i-1$. Each one of these nodes has paths to all other nodes. However, only those paths that go to nodes to the right of $i$ matter because they have to visit $i$ along the way. The nodes to the right of $i$ are $i+1, i+2, \ldots, n$, and if we count them we find they are $n - (i+1) + 1 = n - i$.*

*Taking one node to the left, it has one path that visits $i$ along the way to $i+1$, another path to $i+2$, etc. In total, for this one node, there are $n-i$ paths going through $i$, one for each node to the right of $i$. Since there are $i-1$ nodes to the left of $i$, each having the same number of paths to nodes to the right of $i$, the total number of paths going through $i$ is $(i-1)(n-i)$. Adding to this the paths between $i$ and other nodes, we have*

$$b_i^{(\text{line } n)} = n - 1 + (i-1)(n-i) = -1 + i(n+1) - i^2. \tag{2}$$

*Note that this is the equation of a parabola in the variable $i$. We can plot $b_i$ to see how it behaves along the line (Fig. 3). If $n \gg 1$, we could approximate $b_i$ as a continuous function in the variable $i$, and calculate the location of*

*the maximum betweenness. For those familiar with calculus, this is done by taking the derivative of $b_i$ with respect to $i$ and equating it to zero, to obtain*

$$\frac{d}{di}b_i\Big|_{i=\tilde{\imath}} = 0 \Rightarrow \frac{d}{di}\left[i(n+1) - i^2 - 1\right]\Big|_{i=\tilde{\imath}} = n + 1 - 2\tilde{\imath} = 0 \Rightarrow \tilde{\imath} = \frac{n+1}{2}.$$

*In fact, this answer is pretty much correct regardless of the continuous approximation of calculus because, for a line with an even number of nodes, the two central nodes have equal maximum betweenness, and for a line with an odd number of nodes, the unique central node has maximum betweenness, i.e.*

$$\tilde{\imath} = \begin{cases} (n+1)/2 & n \text{ odd} \\ n/2 \quad \& \quad 1 + n/2 & n \text{ even.} \end{cases} \tag{3}$$

*As a final result, we calculate the actual value of betweenness centrality at this node of maximum betweenness:*

$$b_{\tilde{\imath}}^{(line\ n)} = \begin{cases} -1 + \frac{n+1}{2}(n+1) - \left(\frac{n+1}{2}\right)^2 & = -1 + \frac{(n+1)^2}{4} \quad [odd\ n] \\ -1 + \frac{n}{2}(n+1) - \left(\frac{n}{2}\right)^2 & = -1 + \frac{n(n+2)}{4} \quad [even\ n] \end{cases} \approx \frac{n^2}{4} \tag{4}$$

*where the last value is an excellent approximation for large $n$.*

## 2 A bit of interpretation about betweenness

As we can see from Exmp. 2 above, the maximum betweenness of a line occurs for the node(s) in the middle because these are the nodes through which the most traffic passes (number of paths through the node(s)). In fact, we calculated the *value* of betweenness for the middle node(s) and found that it is well approximated by $n^2/4$. Should we be surprised by this result?

Let us recall that in a network there are potentially as many as $\binom{n}{2} \approx n^2/2$ pairs of nodes structurally connected, and this in turn means there may be an equal number of possible (shortest) paths between them. For the node of maximum betweenness to have a betweenness value of $n^2/4$ is not surprising if this node performs the role of a bridge between two sides of even sizes of a network, which applies for the middle node(s) of the line network. By a *bridge node* we mean a node that, when removed, increases the number of disconnected clusters in the network (such nodes are also called cut vertices). It is noteworthy that when nodes serve as bridges in a network, the calculation of their values of betweenness centrality can be completed mathematically.

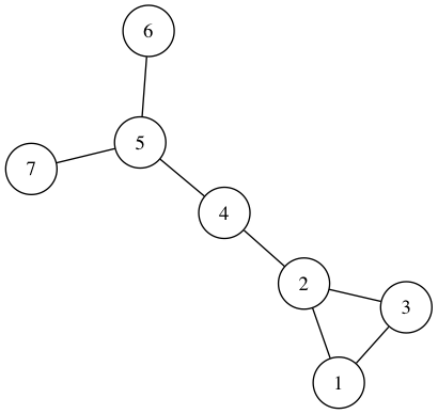


Figure 4: Network for Exmp. 3.

**Example 3.** *Calculate the betweenness centrality of node* 4 *in the network displayed in Fig. 4. The key to the calculation is the fact that the nodes to "one side" of node* 4 *need to go through* 4 *to reach the nodes on the "other side" of* 4*; node* 4 *is a bridge node. To one side of node* 4 *we find nodes* 1,2*, and* 3*, and to the other side of node* 4 *we have nodes* 5,6*, and* 7*. Thus, there are* 3 *nodes either side of* 4*. This realization allows us to write the value of* $b_4$*, the betweenness of node* 4*, as the sum of two contributions*

$$\begin{aligned} b_4 = &\text{ shortest paths from 4 to other nodes} \\ &+ \text{shortest paths from nodes one side of 4 to the other side.} \end{aligned} \quad (5)$$

*Thus, our task is to write down equations for the two terms. The first term is exceedingly simple. It is the number nodes minus* 1 *(since node* 4 *is the origin, it is not counted). The second term can be calculated multiplying the number of nodes to one side of node* 4 *to the number of nodes on the other side. Therefore*

$$b_4 = \underbrace{6}_{\text{paths from 4 to other nodes}} + \underbrace{3 \times 3}_{\text{paths going through node 4}} = 15$$

**Example 4.** *Estimate the value* $n^2/4$ *of maximum betweenness in the line network. This is done in a similar way as in the previous example (Exmp. 3. To the left of the line network there are roughly* $n/2$ *nodes, and the same number to the right. This means that*

$$b_{\tilde{\imath}} \approx \frac{n}{2} \times \frac{n}{2} + n - 1 \approx \frac{n^2}{4}.$$

The maximum betweenness centrality is a quantity of general interest. It serves as an estimate of the maximum traffic any network node may be managing. It depends on three factors, 1) the size of a network, 2) the network structure, and 3) the paths used to connect between nodes (recall the idea of a routing protocol above). A maximum betweenness of the order of $n^2$ is one of the worse situations. However, it is also frequently encountered (see Exmp. 3). Fortunately, in some networks the exponent is less than 2 as a consequence of the network structure. There is considerable research in trying to find ways to reduce the exponent to even lower numbers because this means the amount of traffic of the most loaded node is not as large. This is usually done by choosing paths between nodes that are not the shortest paths, although a price is paid for this in that paths are longer, and more nodes in the network experience traffic similar to those of the most loaded nodes.

## 3 Eigenvector centrality

This is a more subtle measure than the previous one because it does not come directly from something we can count; it depends on an equation that involves the entire adjacency matrix. In other words, it is a global measure. How is it defined?

Let us call $x_i$ the eigenvector centrality of node $i$; the reason for the name will become clear shortly. The starting point for the idea behind $x_i$ is anchored in the social world. Imagine that a person $i$ in a social network has a "relevance" in the network that is due to two factors:

1. the number of individuals person $i$ is connected to, and
2. the "relevance" of those individuals $i$ is connected to.

In other words, this centrality partly measures importance by "who you know".

A final, more technical idea behind $x_i$ is that it depends linearly on the two contributions enumerated above. We can convert these statements into the concrete formula

$$x_i = \beta \sum_{j=1}^{n} a_{ij} x_j, \qquad i = 1, 2, \ldots, n. \tag{6}$$

To see how this relates to the ideas above, note first that the relation is indeed linear as it only depends on the *sum* over $x_j$. Also, it is clear that

if $j$ has a large value of $x_j$ and $i$ and $j$ are connected (and thus $a_{ij} = 1$), then $x_j$ contributes its large importance to the value of $x_i$. Finally, the more connections $i$ has, which means that there are more link indicators $a_{ij}$ equal to 1, the more contributions there are to increase the value of $x_i$.

Well this is progress! We now have a way to related values of $x_i$ for $i = 1, 2, \ldots, n$ with one another. Furthermore, note that Eq. 6 is not just one relation, but it is actually $n$ relations, one for every node $i$. In addition, the constant $\beta$ is the same in all the equations because of a need for consistency among them.

However, given all this information, we still need a concrete way to calculate all the $x_i$, hopefully a method we might be familiar with. To get to this point, let us rewrite Eq. 6 as a matrix equation. First, note that all the $x_i$ can be organized into a column matrix

$$\mathbf{x}^T = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}. \tag{7}$$

The superscript $T$ is a technicality that means that $\mathbf{x}$ is defined as the row vector $(x_1, x_2, \ldots, x_n)$ but to write it as a column vector we have used its *transpose* (you can ignore that $\mathbf{x}$ is a row vector and just think that the relevant quantity is $\mathbf{x}^T$). Now, a column vector is the same as a matrix with a single column. This means that, for instance, at row $i$, the element $x_i$ can also be written as $x_{i,1}$ where the 1 represents the column. However, since there is only the one column, we ignore the 1 in the subindex and end up with Eq. 7.

With this vector defined, we can also write the right-hand-side of Eq. 6 in matrix form by using a matrix product between $\mathbf{x}^T$ and $\mathbf{A}$. Specifically, $\sum_{j=1}^{n} a_{ij}x_j$ is also $\sum_{j=1}^{n} a_{i,j}x_{j,1}$, and by the definition of matrix multiplication this is equal to the row $i$ of $\mathbf{A}$ multiplied element by element with the column vector $\mathbf{x}^T$, and corresponds to the element in row $i$ and column 1 of the result. In other words

$$\sum_{j=1}^{n} a_{ij}x_j = (\mathbf{A}\mathbf{x}^T)_{i,1} \tag{8}$$

and by Eq. 6, this is

$$x_i = \beta(\mathbf{A}\mathbf{x}^T)_{i,1}. \tag{9}$$

Both the right and left hand sides of this expression are simply row $i$ of column vectors. On the left, it is the column vector $\mathbf{x}^T$ and on the right it is

the column vector $\mathbf{A}\mathbf{x}^T$. Therefore, defining $\lambda = 1/\beta$, the equation becomes

$$\lambda \mathbf{x}^T = \mathbf{A}\mathbf{x}^T. \tag{10}$$

For those with experience in linear algebra, you may recognize Eq. 10 as that of an eigenvalue problem. This is an important problem defined on matrices, such as $\mathbf{A}$, that are square (with equal numbers of rows and columns) and invertible (which means it is possible to find a matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$). The kind of solution that the problem has is interesting: the solution consists for several (at most $n$, because of the dimensions $n \times n$ of $\mathbf{A}$) pairs $\lambda_u, \mathbf{x}_u^T$ where $u$ is simply the counter of possible solutions. A given solution with $\lambda_u$ and $\mathbf{x}_u^T$ is called the uth eigenvalue and eigenvector pair.

There are many wonderful and interesting properties associated with this problem that we cannot get into in this presentation. However, for our purposes, we point out that because of $\mathbf{A}$ is made up of real numbers and is symmetric, it means that out of all the eigenvalues, there is one that is real and larger than all the others, and also all the elements of the associated eigenvector are all positive (all the $x_i$). This eigenvalue is called the largest eigenvalue. The associated eigenvector provides us with the centralities we want to find. Note that if we multiply Eq. 10 by a constant number on both sides, this number can be absorbed into $\mathbf{x}^T$ and thus means that eigenvectors are specified to within a constant factor. Therefore, it is typical to impose an extra condition on the vector such as $\sum_{i=1}^{n} x_i^2 = 1$.

We are now prepared to introduce the following definition

**Definition 2.** *The* **eigenvector centrality** $x_i$ **of node** $i$ *is a relevance score for a node in a network with adjacency matrix* $\mathbf{A}$ *that is given by the* $i$*-th element of the eigenvector* $\mathbf{x}$ *associated with the largest eigenvalue* $\lambda$ *of Eq. 10. It is common, although not required, to normalize the eigenvectors so that the overall magnitude of* $\mathbf{x}$ *is equal to 1.*

To actually extract numbers from Eq. 10, we usually employ computational (numerical) methods. The solution of the equation can only be done by hand for small adjacency matrices, and in this case one begins by finding the eigenvalues and with each one of those we can find the corresponding eigenvectors. Let us now show a small example that can be done by hand, and offer computational tools below (**this is basically a reminder of calculating eigenvalues and eigenvectors**.)

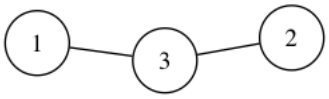


Figure 5: A v-shape network.

**Example 5.** *Let us consider the v-shape network, shown in Fig. 5. The adjacency matrix for the network is*

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

*To determine the centralities in this problem, we must solve for the elements of* $\mathbf{X}$ *by using Eq. 7, which gives in our example*

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \lambda \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

*To solve for* $\{x_a, \ldots, x_d\}$*, we must solve the* **eigenvalue problem**. *There is a procedure to follow. First, write the equation in the form*

$$\left[ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} - \lambda \mathbf{I}_3 \right] \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\Longrightarrow \left[ \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right] \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

*where we have used the identity matrix* $\mathbf{I}_3$ *which when multiplied to the right by a matrix* $\mathbf{B}$ *with dimensions* $3 \times c$ *for any* $c > 0$ *results in* $\mathbf{B}$ *again. Thus,* $\mathbf{I}_3 \mathbf{x}^T = \mathbf{x}^T$.

*The matrices in the square brackets can then be added*

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -\lambda & 0 & 1 \\ 0 & -\lambda & 1 \\ 1 & 1 & -\lambda \end{pmatrix}.$$

*We then calculate the determinant of this resulting matrix and make it equal*

*to zero to solve for the value(s) of $\lambda$ as follows:*

$$\det\begin{pmatrix} -\lambda & 0 & 1 \\ 0 & -\lambda & 1 \\ 1 & 1 & -\lambda \end{pmatrix} = -\lambda^3 - (-\lambda - \lambda) = 0$$

$$\implies -\lambda(\lambda^2 - 2) = 0 \implies \lambda = \begin{cases} 0 \\ +\sqrt{2} \\ -\sqrt{2}. \end{cases}$$

*The largest eigenvalue is associated with an eigenvector that is made up of all positive values $x_i$. Thus, we reinsert $\sqrt{2}$ into the original matrix equation of our problem to obtain*

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \sqrt{2} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

*which can then be solved for $x_1, x_2, x_3$. Note that as a system of equations, this takes the form*

$$\begin{cases} x_3 & = & \sqrt{2}x_1 \\ x_3 & = & \sqrt{2}x_2 \\ x_1 + x_2 & = & \sqrt{2}x_3. \end{cases}$$

*From these equations, it is clear that $x_1 = x_2 = x_3/\sqrt{2}$. The third equation is not going to give us new information.*

*However, at this point we are ready to state some conclusions. First, $x_3 > x_1 = x_2$. We have acknowledged that it is the relative ranking between $x$ values that matters. For the v-shape $x_3 = \sqrt{2}x_2 = \sqrt{2}x_1$. Another feature to note is that, as expected by symmetry of node label swap between 1 and 2, it is clear that both of them perform the same function and thus have the same eigenvector centrality; this matches our calculation.*

*Finally, in many cases to give concrete numbers to $\{x_1, x_2, x_3\}$ they are required to satisfy*

$$x_1^2 + x_2^2 + x_3^2 = 1. \tag{11}$$

*This, however, does not change the nature of the result. In our case,*

$$\left(\frac{x_3}{\sqrt{2}}\right)^2 + \left(\frac{x_3}{\sqrt{2}}\right)^2 + x_3^2 = 1 \implies x_3^2\left(\frac{1}{2} + \frac{1}{2} + 1\right) = 1 \implies \begin{cases} x_1 & = & \frac{1}{2} \\ x_2 & = & \frac{1}{2} \\ x_3 & = & \frac{1}{\sqrt{2}}. \end{cases}$$

## 4 A bit of interpretation of eigenvector centrality

Note that Eq. 10 determines, up to an arbitrary constant, the values of eigenvector centrality. This means a few things.

First, the exact value of each $x_i$ is not in itself relevant. That is because, as explained, Eq. 10 can be modified by multiplying a constant $c$ throughout and the equation still preserves the same form, except that now $\mathbf{x}^T$ is a quantity $c\mathbf{x}^T$ with a new scale.

What this means is that the relevant aspect of eigenvector centrality is the *comparative values* of the nodes. In other words, it matters more which node has the largest $x$, and not the number $x$ in itself. Thus, to use eigenvector centrality in practice, we take the numbers $x_i$ for all the nodes and then rank-order the nodes by values of $x$. The most important node $q$ has the largest $x_q$ value, whereas the least important node $r$ has the smallest $x_r$ value.

Another important observation about Eq. 10 is that the adjacency matrix $\mathbf{A}$ for the network is the only information needed to determine the $x_i$. If we think about it for a moment, this makes a lot of sense: the values of $x_i$ reflect the relevance of nodes in a specific network, and therefore we need to input network information to find these values. Note, however, that using $\mathbf{A}$ does have to be the only way to input network information. There are other relations that look similar to Eq. 10 that manage to emphasize different aspects of the network structure to calculate node importance. We mention some of those below (Sec. 5). An interesting one is called *PageRank* centrality, the quantity that Google used to measure to determine which websites to show first as the result of a search (their algorithms have changed over time, but PageRank was the one that greatly contributed to their tremendous success and current standing as *the* search engine of the Web).

A more technical interpretation of these results is beyond the scope of our treatment, but it is at least interesting to look at eigenvector centrality as the simplest example of linear-algebra-based measures for node importance.

## 5 Other measures of centrality

Let us name some of the more important ones.

### 5.1 Closeness centrality

For every cluster $c$ in a network $G$, we can calculate the shortest path histogram $H(\ell|i)$ of every node $i \in c$ (every node $i$ that belongs to the cluster $c$), and from this histogram calculate the average shortest path distance from $i$ to all other nodes in $c$. Mathematically, ,

$$\langle \ell_i \rangle = \frac{\sum_\ell \ell H(\ell|i)}{\sum_\ell H(\ell|i)} \tag{12}$$

This quantity, $\langle \ell_i \rangle$ can also be used to define a **closeness centrality**, which is an indication of how "quickly" one can reach other nodes from the specific node. In this case, *small* closeness centrality is equivalent to greater relevance. One can also define the mathematical reciprocal for this quantity, that is $1/\langle \ell_i \rangle$ which then makes the closest node have the largest $1/\langle \ell_i \rangle$, but this is just a matter of preference on how to state the measure of importance.

### 5.2 PageRank centrality

For a moment, we suspend our ban on directed networks to explain this very relevant centrality. Let us give a little background: when you go to a website (imagine something like `www.cnn.com`), this site has bits of text that one can click on (technically called hypertext) that take us to other websites. Conversely, other websites point to `www.cnn.com` because it is a well-known source of information. Note that websites do not have to point at each other, and therefore links here are directed.

When we search for information on the web, we need to figure out what websites are useful to us and which aren't. One of the key ideas that made the search engine Google successful was the realization that websites that are pointed to very often are likely to be more relevant that those that are not.

PageRank exploits this idea of popularity being representative of usefulness. The algorithm generates a score $r$ for each website on the basis of a combination of features. To calculate the score of page $i$, the algorithm basically simulates a walk along websites. The equation is given by

$$r_i = \alpha \sum_{j=1}^{n} a_{j,i} \frac{r_j}{k_j} + \frac{1-\alpha}{n}, \tag{13}$$

where $k_j$ is the degree of node $j$. Let us explain the expression: on the right-hand-side, the sum goes over each node $j$ that has a connection pointing from

$j$ to $i$ and contributes $\alpha r_j/k_j$ to $r_i$; the other contribution is a generic term $(1-\alpha)/n$. Now, what does this all mean? The idea of the ratio $r_j/k_j$ is that if a "walker" is located a node $j$, it has a probability $1/k_j$ to pick $i$ as the neighbor to go to. This is a bit like saying that somebody looking at website $j$ has a likelihood $1/k_j$ to jump to any of the websites that $j$ points to, one of them being $i$. The $\alpha$ is a weight parameter that balances the likelihood that one arrives to $i$ from one of its neighbors or from anywhere in the network, which is expressed by the complement of that, given by $(1-\alpha)/n$.

The ultimate solution for the values of $r_i$ for all $i$ are the PageRank scores of the nodes in the network.

# 6 Centralization

There is an additional notion that applies to an entire network $G$, called centralization. It requires:

1. a choice for a centrality $C_i$ for each node $i$ of the network,
2. the determination of all the values of $C_i$ and the identification of the largest $C$ among the $C_i$, a quantity we call $C^*$,
3. the result of the calculation
$$N(G) = \sum_{i \in G} \left(C^* - C_i\right), \tag{14}$$
4. the determination of a network $\tilde{G}$ with *the same number of nodes* $n$ as $G$ that leads to the maximum possible value of $N$, called $\tilde{N}$ (in other words, $\tilde{N}$ is calculated on the basis of $\tilde{G}$).

Centralization is then
$$\mathcal{C} = \frac{N(G)}{\tilde{N}} = \frac{N(G)}{N(\tilde{G})}. \tag{15}$$

In many cases, $\tilde{G}$ is the star network of $n$ nodes, where one node is the hub and there are $n-1$ satellite nodes all connected only to the hub.

# 7 Computational Tools

To calculate centralities, there is an assortment of algorithms proportional in numbers to the actual distinct centrality measures out there. Constructing

some of these algorithms from scratch is in fact one of the key abilities of a network scientist. One caveat I should mention is that those algorithms that involve the calculation of linear algebra quantities such as the determination of eigenvalues and vectors is an extremely well developed and specialized field, and in this case, it is best to rely on well-established methods.

Determining degree and closeness centralities can be done on the basis of algorithms and methods we have already studied, and thus are not worth revisiting. On the other hand, bare betweenness centrality is new to us, and in this section we build the pseudocode for it, and also present our own algorithm and those of `networkx` that address some of the betweenness definitions commonly in use.

## 7.1 Bare betweenness centrality: pseudocode and code

In order to calculate bare betweenness, the quantity to determine is the number of shortest paths visiting a node. This is a quantity that is not fundamentally difficult to calculate, although doing it efficiently can require a bit of cunning.

### 7.1.1 Bare betweenness due to shortest paths starting from a single node

The fundamental tool to calculate betweenness is the method and associated algorithm to determine shortest paths in a network. This algorithm is called Dijkstra's algorithm and is a version of so-called breadth-first-search (BFS). Breadth-first search is an algorithm of great utility. If you use SatNavs to drive from place to place, you are invoking BFS. The most common use of BFS requires we specify a starting location and then the algorithm exhaustively identifies how one can travel from that starting location to all other locations that are structurally connected.

A intuitive description of BFS is as follows: from an origin node (call it $o$), BFS identifies all the neighbors of $o$ (nodes such that $a_{o,i} = 1$) and marks them as being at distance $\ell = 1$ from $o$. This forms a group of nodes $\mathcal{S}_{\ell=1}^{(o)}$ that we refer to as shell 1 from origin $o$ (as indicated below, we have good reasons to generally omit the superindex $o$, but we leave it for now). As an auxiliary concept, we create $\mathcal{S}_{\ell=0} = \{o\}$, a set that contains only the origin node and is labelled as shell 0. Then, from the nodes in $\mathcal{S}_{\ell=1}^{(o)}$, BFS identifies out of the node neighbors of each $i \in \mathcal{S}_{\ell=1}^{(o)}$ those that do not belong to either $\mathcal{S}_{\ell=0}^{(o)}$ or $\mathcal{S}_{\ell=1}^{(o)}$. In other words, there is a set of nodes $j$ such that

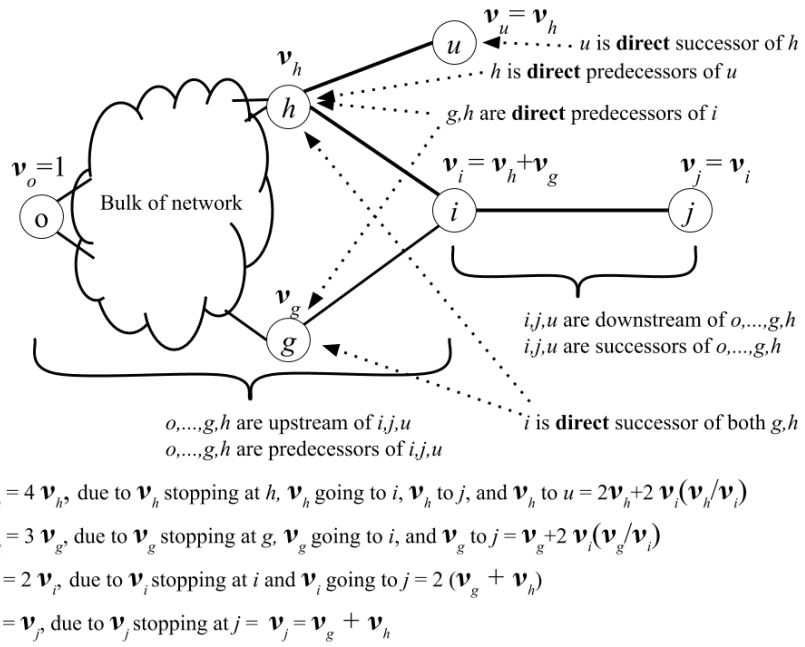


Figure 6: Quantities involved in the **calculation of the betweennesses due to origin** $o$. Illustration of the situation and terms associated with the algorithm for bare betweenness

$a_{ji} = 1$ if $i \in \mathcal{S}^{(o)}_{\ell=1}$ but $j \notin \mathcal{S}^{(o)}_{\ell=1}$ and $j \notin \mathcal{S}^{(o)}_{\ell=0}$. These nodes constitute $\mathcal{S}^{(o)}_{\ell=2}$, shell 2, and are characterized by their shortest distance to $o$ being $\ell = 2$. In other words, BFS progressively moves away from $o$, without backtracking and in the process identifying for every node its shortest distance from $o$. In addition, each step forward from shell $\ell$ to $\ell + 1$ allows the algorithm to identify for every node $i$ all the various shortest paths (numbered $\nu_i$ and explain below) that can be used to reach $i$.

To determine bare betweenness, we need to perform BFS departing from an origin, for instance $o$, until we find shortest paths to all the structurally connected nodes to that origin (the forward BFS), and subsequently perform a reverse BFS that departs from the farthest nodes and systematically moves back to $o$. By doing this, the algorithm first finds the shortest paths from $o$ (during the forward movement), and then by going in reverse it counts all the nodes that those shortest paths reached, providing a way to determine the betweenness of each node due to origin $o$. Nodes that have been reached after going through a node, say $i$, are said to be downstream of $i$; these

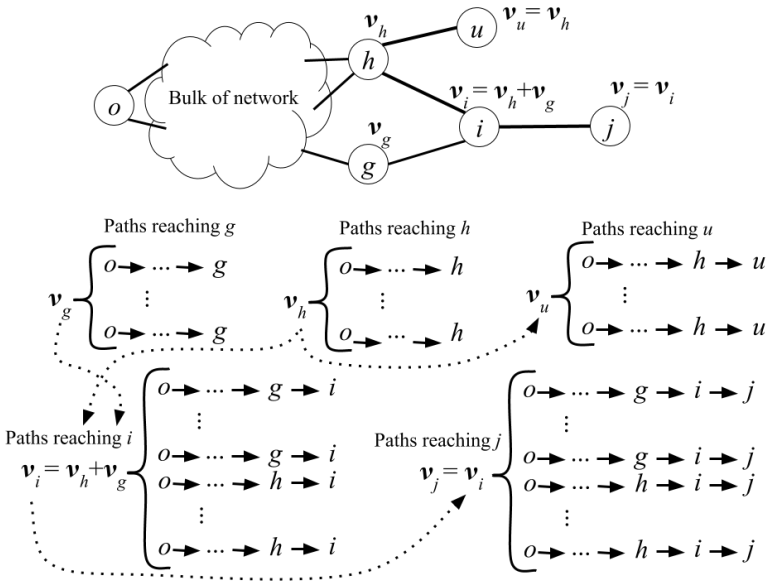


Figure 7: Further details on the bare betweenness of a set of nodes in a network. The illustration highlights the way in which the multiplicity of paths into a node $i$ comes into that node through its predecessors, and is also transferred to other nodes that are successors of $i$.

nodes are also called *successors* of $i$. On the other hand, if node $h$ was crossed along the way to reaching node $i$, then $h$ is upstream of $i$; we also call $h$ a *predecessor* of $i$ (an illustration defining these names can be found in Fig. 6).

To compute the bare betweenness of a node, we need to consider two important effects. First, any node $i$ that can be reached from the origin via multiple shortest paths has at least as many units of bare betweenness. Thus, if $i$ can be reached via $\nu_i$ shortests paths, then $b_i$ contains at least $\nu_i$ units of betweenness. We call $\nu_i$ the *multiplicity* of shortest paths to $i$[2]. Second, since nodes downstream of a node can be reached from any of the paths reaching the latter node, *any node that is a successor to node $i$ contributes $\nu_i$ units to $b_i$*. Related to this, if $i$ has multiplicity $\nu_i$, then the multiplicity $\nu_j$ of successor node $j$ is equal or larger than $\nu_i$. To combine

[2]Since the multiplicity depends on the origin $o$, in principle we should write something like $\nu_i^{(o)}$ highlighting the fact that this multiplicity is influenced by the origin we use. However, since we always construct betweenness one origin at a time, in our discussion, we only need to talk about $\nu_i$ because $o$ is implied, thus simplifying the notation.

these two contributions into the actual calculation of each $b_i$, we now discuss the details. We illustrate the situation via Figs. 6 and 7.

First, let us clear something up: $\nu_i$ is *not* usually $b_i$ because it only takes into account the paths that departed from origin $o$ and reach $i$, but it does not take into account the paths that go *through* $i$ further into its successors in the forward BFS. In other words, $\nu_i$ *is* the contribution to $b_i$ due to paths departing $o$ and stopping at $i$. $\nu_i$ is determined while BFS runs *forward*, and it is equal to the number of paths that depart $o$ and arrive simultaneously at $i$. The remaining contributions to $b_i$ are a function of the number of successors to $i$, and will be determined from continuing BFS until all structurally connected nodes are reached. In fact, as we will illustrate below, it turns out that if the number of successors of $i$ is $s_i$, then $b_i = \nu_i \times (s_i + 1)$. It is worth noting though that for nodes where the forward BFS stops because there are no other downstream nodes, the betweennesses of those nodes *are* equal to their multiplicities since shortest paths end at these nodes and there are no other paths that go through them towards other nodes.

The algorithm reverses course (reverse BFS) moving back towards the origin by starting simultaneously from all the nodes that belong to the farthest shell from $o$. Say that one of those nodes, called $j$, has been reached in $\nu_j$ ways by the forward BFS, and has one direct predecessor $i$. Then $\nu_j = \nu_i$[3]. Now, since $\nu_i$ paths reach $i$ and a further $\nu_j = \nu_i$ paths continue onto $j$, it means that there are $\nu_i + \nu_j = \nu_i + \nu_i = 2\nu_i$ paths visiting $i$, those that stay at $i$ and those that go through to $j$. This translates into $b_i = 2\nu_i$.

Suppose further that $i$ has two direct predecessors $g$ and $h$, with respective multiplicities $\nu_g$ and $\nu_h$. These two multiplicities are responsible for $\nu_i$, i.e. $\nu_i = \nu_g + \nu_h$. Now, for instance, $\nu_h$ means that there are exactly $\nu_h$

[3]One way to visualize this is by constructing the paths from $o$ to $i$ and then to $j$ explicitly. Thus, $\nu_i$ paths means that

$$\nu_i \text{ paths} \begin{cases} o \to \cdots \to i \\ \quad \vdots \\ o \to \cdots \to i \end{cases} \tag{16}$$

Then, the paths to $j$ (where $i$ is its only successor) are the same as those to $i$ with the extra step to reach $j$, or

$$\nu_i \text{ paths} \begin{cases} o \to \cdots \to i \to j \\ \quad \vdots \\ o \to \cdots \to i \to j \end{cases} \tag{17}$$

illustrating why in this case $\nu_j = \nu_i$. Note that if $j$ has more than 1 predecessor, then there will need to be additional paths and in that case $\nu_j > \nu_i$.

shortest paths that have $h$ as their final node. On the other hand, there are also $\nu_h$ shortest paths that continue onto $i$ (representing the fraction $\nu_h/(\nu_h+\nu_g) = \nu_h/\nu_i$ of the total multiplicity of $i$) and an extra $\nu_h$ paths continuing from $i$ to $j$ (contributing the fraction $\nu_h/\nu_i$ to the total multiplicity of $j$). So at this point, we have counted $3\nu_h$ paths through $h$. But notice that in Figs. 6 and 7 there is a node $u$ that is a direct successor to $h$ with no other predecessors or successors itself; its multiplicity is $\nu_u = \nu_h$, i.e., entirely contributed by $h$, and it means that there are an extra $\nu_h$ paths going through $h$. Thus, in total there are $4\nu_h$ paths visiting $h$, leading to $b_h = 4\nu_h$. Incidentally, $b_u = \nu_u = \nu_h$.

Separately, with regards to $g$, there are $\nu_g$ paths stopping at $g$, $\nu_g$ going through to $i$ (contributing the fraction $\nu_g/\nu_i$ of the multiplicity of $i$), and another $\nu_g$ going through to $j$ (contributing $\nu_g/\nu_i$ of the multiplicity of $j$). These contributions give a total of $3\nu_g$ paths visiting $g$ and therefore $b_g = 3\nu_g$.

We may read the previous description as suggesting that we can determine betweenness merely through the forward BFS. In fact that may in principle be true, but it leads to a slower and more memory hungry algorithm. Note that as BFS is moving forward, it creates for every node $z$ its multiplicity $\nu_z$. But at that point the forward BFS still does not know the number of successor nodes to node $z$. One could naively decide to start counting successors to $z$ once it has been reached. But in order to have all the betweennesses, one would have to track successors to *all* the nodes visited, and this is computationally very costly.

The efficient solution is to walk in reverse along each chain of successors, which can be used to determine bare betweennesses without the need to update an increasing number of the nodes. One merely needs to update the nodes being visited at any given time by the BFS running in reverse. The key to this strategy is to use the relations between multiplicities and betweennesses of nodes that are direct successors/predecessors of each other. Note, for example, that

$$b_g = \underbrace{\nu_g}_{\text{paths to } g} + \underbrace{\nu_g}_{\text{to } i} + \underbrace{\nu_g}_{\text{to } h} = \nu_g + 2\nu_g$$

$$= \nu_g + \underbrace{2\nu_i}_{=b_i} \times \underbrace{\left(\frac{\nu_g}{\nu_i}\right)}_{\text{part of } \nu_i \text{ due to } \nu_g} = \nu_g + b_i \times \frac{\nu_g}{\nu_i}. \quad (18)$$

A similar thing happens with $h$, which can be written as

$$b_h = \underbrace{\nu_g}_{\text{paths to } h} + \underbrace{\nu_g}_{\text{to } u=b_u} + \underbrace{\nu_g}_{\text{to } i} + \underbrace{\nu_g}_{\text{to } j} = \nu_h + b_u + 2\nu_h$$
$$= \nu_h + b_u \times \underbrace{\frac{\nu_h}{\nu_u}}_{=1} + 2\nu_i \times \frac{\nu_h}{\nu_i} = \nu_h + b_u \times \frac{\nu_h}{\nu_u} + b_i \times \frac{\nu_h}{\nu_i} \quad (19)$$

In general, for a node $z$ with direct successor nodes $y_1, y_2, \ldots, y_q$, its bare betweenness is given by

$$b_z = \nu_z + \sum_{v=1}^{q} b_{y_v} \frac{\nu_z}{\nu_{y_v}} = \nu_z \left(1 + \sum_{v=1}^{q} \frac{b_{j_v}}{\nu_{j_v}}\right) \qquad \text{(for a given origin)}. \quad (20)$$

The forward plus reverse BFS algorithm explained next works by using this equation, filling in the values that go into it and determining all the $b_i$[4].

As a final comment before ending this section, we remind the reader that up to this point we have described the calculation of $b_i$ **due to a specific origin** $o$. We still need to go through all origins to obtain a total betweenness. Therefore, the reader should be careful at this point when trying to compare the results of $b_i$ from Eq. 20 to those of another algorithm to determine $b_i$ (such as inspection as we have used in Exmp. 1). To make the right comparison, the reader would need to only use paths departing from the fixed origin for which Eq. 20 is being applied.

### 7.1.2 Overall bare betweenness for each node

The final step in determining the overall bare betweenness of every node is to go through all origins (all nodes in the network) and construct the betweennesses in the same way. The overall value of $b_i$ is the sum of all such contributions of betweenness divided by 2 since going through all origins, we double count betweenness for each node.

### 7.1.3 Bare betweenness centrality pseudocode

We now have all the information needed to make the pseudocode for betweenness centrality. What we want to do is to perform a loop over every node in the network acting as an origin for a BFS (breadth-first search)

[4]For those readers that are curious about some of the relations going on in the background between bare betweenness and the adjacency matrix, note that each $\nu_i$ is equal to the value of $\mathbf{A}_{o,i}^{\ell_{o,i}}$ where $\ell_{o,i}$ is the shortest path length between $o$ and $i$.

that has been modified to track predecessors and multiplicities. In that same loop step, once BFS runs and predecessors for all nodes are identified, we walk the shells *backwards* so that we count the number of nodes that are successors of other nodes, along with their multiplicities, which gives us the units of bare betweenness centrality that originate for that given origin node. Once the loop step ends, the next step starts BFS from a different origin and calculates the additional units of betweenness generated on each node by this new origin. When the loop finishes, all the betweennesses are determined. Let us first show the modified BFS, which we call BC1, seen in Alg. 21. Let us describe the differences with respect to BFS. First, note that we are no longer saving the length of the shortest path with respect to $i$ for every node because we do not need to report this back at the end of the algorithm. On the other hand, we have introduced three new objects which we need to behave as `python` dictionaries, called $p, z$, and $m$. In $p$, we store the predecessor(s) of a node, so that if $j$ is the node and its predecessor is $i$, then $i \in p(j)$. Note that $p(j)$ can have more than one predecessor (we skipped mentioning this before for simplicity), a situation that happens if a node $j$ in shell $\ell$ could legitimately be reached from nodes $i_1, i_2, \ldots$ all in shell $\ell - 1$. Because of this, $p(j)$ is populated by a list of objects, all of which are predecessors of $j$; in many cases, $p(j)$ contains more than one element. The dictionary $z(\ell)$ keeps a list of the nodes located in shell $\ell$. This is important because when we start to move in reverse, we want to do it in order from the most distant nodes; this allows us to count all the nodes that are in the succession of each node. Finally, $m(i)$ tracks the multiplicity of $i$.

When BC1 ends, it returns the betweenness centralities due to origin $o$. To obtain a full count of the betweenness centrality of all the nodes due to the entire network, we must then go through every possible origin $o$ and add up their contributions to each $i$. There is a small caveat: by starting from every possible origin we double count paths and therefore, the final result requires dividing by $2^5$. In summary, to obtain the full $b(i)$ due to all possible unique shortest paths in the network, we perform the loop in Alg. 22.

### 7.1.4 Bare betweenness centrality code and examples

Now, the code that emerges from the previous pseudocode, written in `python`, is as follows

[5]Some definitions of betweenness centrality accept the double counting as part of the definition. In those cases, there is no need to correct for double counting.

**BC1**$(G, o)$ :
  $\mathcal{V} = \{\}$
  $\mathcal{A} = \{o\}$
  $\ell = 0$
  $p(o) = []$
  $m(o) = 1$
  $z(\ell) = [o]$
  **while** $|\mathcal{A}| > 0$ :
  |  $\ell = \ell + 1$
  |  $\mathcal{T}_{\mathcal{A}} = \{\}$
  |  **for** $i$ **in** $\mathcal{A}$ :
  |  |  **for** $j$ **in** $\{1, \ldots, n\}$ :
  |  |  |  **if** $a_{ij} = 1$
  |  |  |  |  **if** $(j$ **not in** $\mathcal{V})$ && $(j$ **not in** $\mathcal{A})$ :
  |  |  |  |  |  **if** $j$ not in $\mathcal{T}_{\mathcal{A}}$ :
  |  |  |  |  |  |  $\mathcal{T}_{\mathcal{A}} = \mathcal{T}_{\mathcal{A}} + \{j\}$
  |  |  |  |  |  |  $z(\ell) = z(\ell) \cup \{j\}$
  |  |  |  |  |  $p(j) = p(j) \cup \{i\}$
  |  |  |  |  |  $m(j) = m(j) + m(i)$
  |  |  $\mathcal{V} = \mathcal{V} \cup \{i\}$
  |  $\mathcal{A} = \mathcal{T}_{\mathcal{A}}$
  $\ell_f = \ell - 1$
  **for** $i$ **in** $\{1, \ldots, n\}$ :
  |  $b(i) = 0$
  **for** $l = \ell_f, 1, -1$ :
  |  **for** $i$ **in** $z(l)$ :
  |  |  $b(i) = b(i) + m(i)$
  |  |  **for** $j$ **in** $p(i)$ :
  |  |  |  $b(j) = b(j) + m(j) \times b(i)/m(i)$
  **return**$(b)$

(21)

---

```
def BC1(G,o):
    A=[o]
    V=[]
    l=0
    p={}
    m={}
    z={}
    p[o]=[]
```

$$
\begin{array}{l}
\mathbf{BC}(G): \\
\quad \textbf{for } i \textbf{ in } \{1,\ldots,n\}: \\
\quad | \quad B(i) = 0 \\
\quad \textbf{for } o \textbf{ in } \{1,\ldots,n\}: \\
\quad | \quad b = BC1(G,o) \\
\quad | \quad \textbf{for } i \textbf{ in } \{1,\ldots,n\}: \\
\quad | \quad | \quad \textbf{if } i \textbf{ in } b: \\
\quad | \quad | \quad | \quad B(i) = B(i) + b(i) \\
\quad \textbf{for } i \textbf{ in } \{1,\ldots,n\}: \\
\quad | \quad B(i) = B(i)/2 \\
\quad \textbf{return}(B)
\end{array}
\tag{22}
$$

```
    m[o]=1
    z[l]=[o]
    while len(A)>0:
#       print len(A)
        l=l+1
        nA=[]
        for i in A:
            for j in G.neighbors(i):
                if (j not in V) and (j not in A):
                    if  j not in nA:
                        nA.append(j)
                        z[l]=z.get(l,[])
                        z[l].append(j)
                    p[j]=p.get(j,[])
                    p[j].append(i)
                    m[j]=m.get(j,0)
                    m[j]=m[j]+m[i]
            V.append(i)
        A=nA
    lf=l-1
    b={}
    for i in G.nodes():
        b[i]=0
    for l in range(lf,0,-1):
        for i in z[l]:
            b[i]=b[i]+m[i]
            for j in p[i]:
                b[j]=b[j]+m[j]*b[i]/m[i]
    return(b)
```

```
def BC(G):
    B={}
    for i in G.nodes():
        B[i]=0
    for o in G.nodes():
        b=BC1(G,o)
        for i in b.keys():
            B[i]=B[i]+b[i]
    for i in G.nodes():
        B[i]=B[i]/2
    return(B)
```

To make use of the algorithm, we can use it with a network. For instance, we could check our work for the line network above. First, in a `python` session, we define the line network `G` (in the example we do it with 5 nodes), and then call the betweenness centrality algorithm

```
>>> import networkx as nx
>>> from betweenness import *
>>> G=nx.Graph()
>>> n=5
>>> for i in range(n-1):
...     G.add_edge(i,i+1)
...
>>> bg=BC(G)
>>> bg
{0: 4, 1: 7, 2: 8, 3: 7, 4: 4}
```

It is not difficult to check that these values of betweenness are correct: 1) for node 0 there are 4 shortest paths, one for each of the other nodes, 2) for node 1 there are 4 shortest paths to the other nodes *plus* three paths that depart from node 0 and go to nodes 2, 3, and 4, respectively, 3) for node 2 there are the 4 paths to the other nodes *plus* 4 paths passing through, one between 0 and 3, one between 0 and 4, one between 1 and 3, and one between 1 and 4. Finally, by symmetry it is clear that node 3 should have the same betweenness centrality as node 1, and node 4 the same betweenness centrality as node 0.

A more stringent test can be performed by trying to repeat the result calculated mathematically in Exmp. 2 for a line network of $n = 100$ nodes. Therefore, we first calculate the betweenness values, and then proceed to plot them and compare them to the example.

```
>>> import matplotlib as plt
>>> H=nx.Graph()
```

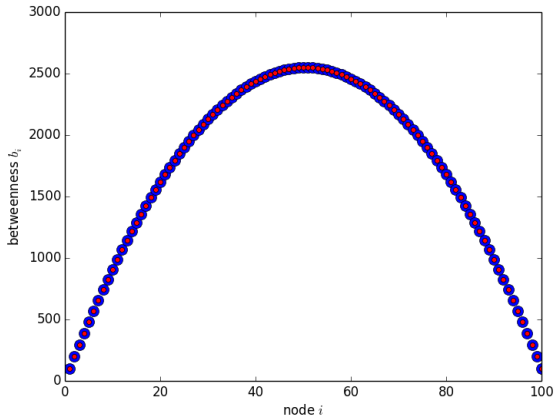


Figure 8: Plot for the theoretically (red) and computationally (blue) calculated betweenness $b_i$ for all the nodes of a line network of $n = 100$. The agreement between theory and computation is clear.

```
>>> n=100
>>> for i in range(1,n):
...     H.add_edge(i,i+1)
...
>>> bh=BC(H)
>>> bt={}
>>> for i in range(1,n+1):
...     bt[i]=i*(n+1)-i**2-1
...
>>> plt.xlabel('node $i$')
>>> plt.ylabel('betweenness $b_i$')
>>> plt.plot(bh.keys(),bh.values(),'o',markersize=10)
>>> plt.plot(bt.keys(),bt.values(),'or',markersize=5)
>>> plt.show()
```

The code produces the plot in Fig. 8.

## 7.2 Eigenvector centrality

As mentioned earlier, because eigenvector centrality requires the application of linear algebra routines (which are a very specialized area), we simply make use of those routines directly. The `python` library that is of interest to us for this is `scipy.linalg`, the linear algebra sub-library for `scipy`.

To illustrate the use of `scipy.linalg` in combination with `networkx`,

let us return to Exmp. 5. We define the network of interest

```
>>> U=nx.Graph()
>>> U.add_edge(1,3)
>>> U.add_edge(2,3)
>>> Au=nx.adjacency_matrix(U).toarray()
```

Note that we have directly saved `Au` in the generic `array` format for `scipy`. That is because `scipy.linalg` does not accept the default format used by `networkx` for adjacency matrices.

The linear algebra function we need to apply is called `eig`. This is a function that determines both the eigenvalues and eigenvectors of a square matrix. The results that come out of `eig` are organized in a certain way explained below. The eigenvectors have a normalized length, i.e. they satisfy the condition Eq. 11. The eiganvalues and eigenvectors can be complex numbers, which means that they sometimes show an imaginary part with `j` representing the imaginary unit (e.g. $a + bj$). Applied to `Au` above, we obtain

```
>>> R=lin.eig(Au)
>>> R[0]
array([ -1.41421356e+00+0.j,   9.77950360e-17+0.j,   1.41421356
    e+00+0.j])
>>> R[1]
array([[  5.00000000e-01,  -7.07106781e-01,  -5.00000000e-01],
       [  5.00000000e-01,   7.07106781e-01,  -5.00000000e-01],
       [ -7.07106781e-01,  -9.02056208e-17,  -7.07106781e-01]])
```

Let us study these answers in comparison with those in Exmp. 5. First, note the organization: the first element of the results, stored in `R`, is a one dimensional array that contains all the eigenvalues. We can see this by doing a bit of arithmetic. In Exmp. 5, we calculated the eigenvalues to be $\sqrt{2}, 0, -\sqrt{2}$. Now, with the use of a calculator we can see that

$$\sqrt{2} = 1.41421356\ldots$$

which says that two of the eigenvalues calculated by `eig` correspond to $\pm\sqrt{2}$. The remaining eigenvalue calculated numerically is `9.77950360e-17` which is an exceedingly small number; it is in fact the corresponding result for the eigenvalue 0 to the limit of numerical precision of the computer.

The second element of `R` contains the eigenvectors. They are organized in a matrix, with each eigenvector occupying a *column*. The order in which the eigenvalues and eigenvectors are organized is consistent: the $i$-th eigenvalue

in the 1-dimensional array has a corresponding eigenvector located in the $i$-th column of the matrix of eigenvectors. In the results above, the largest eigenvalue `1.41421356e+00+0.j` is the last element in the 1-dimensional array (with index 2) and therefore the last column of the matrix is the corresponding eigenvector. To illustrate, see the following code

```
>>> R[0][2]
(1.4142135623730949+0j)
>>> R[1][:,2]
array([-0.5        , -0.5        , -0.70710678])
```

Looking at the eigenvector, we see that all the values are negative. This is not a problem because in the eigenvalue problem the eigenvectors can be multiplied by a constant and remain valid eigenvectors. Thus, we can multiply the eigenvector by $-1$ to obtain the correct result (note that the other eigenvectors have a combination of positive and negative elements, which means that if we multiply by $-1$ we would still have positive and negative elements, an incompatible result with the condition that all centralities must be positive). The first and second elements of the eigenvector are `0.5`, consistent with Exmp. 5 which says that $x_1 = x_2 = 1/2$. The centrality for 3, $x_3$ is given by `0.70710678` which is, up to numerical precision, equal to $1/\sqrt{2} = 0.70710678\ldots$

The location of the largest eigenvalue in the 1-dimensional array of the solution provided by `eig` is not fixed. Thus, we need to determine its location in order to find the corresponding eigenvector. For a small network this can be done by visual inspection, but large networks require an algorithm. For this purpose, we can write

```
>>> imax=0
>>> evmax=R[0][0]
>>> for i in range(1,len(R[0])):
...     ev=R[0][i]
...     if ev>evmax:
...         evmax=ev
...         imax=i
...
>>> evmax
(1.4142135623730949+0j)
>>> imax
2
>>> R[1][:,imax]
array([-0.5        , -0.5        , -0.70710678])
```

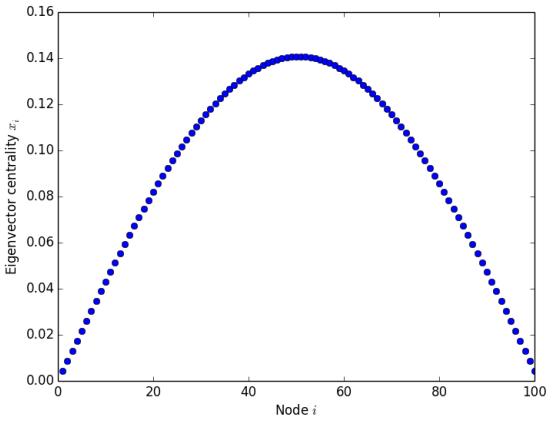


Figure 9: Eigenvector centrality values $x_i$ for the nodes of an $n$-line network. These centralities, their absolute values not withstanding, behave in a similar way as betweenness centralities which basically display a parabola with a maximum around the middle of this network.

The same code allows us to deal with larger problems. For instance, let us revisit the line network with 100 nodes that we had stored as `H`. Following the same steps as above, we find

```
>>> Ah=nx.adjacency_matrix(H).toarray()
>>> Rh=lin.eig(Ah)
>>> imax=0
>>> evmax=Rh[0][0]
>>> for i in range(1,len(Rh[0])):
...     ev=Rh[0][i]
...     if ev>evmax:
...         evmax=ev
...         imax=i
...
>>> imax
35
>>> evmax
(1.9990325645839744+0j)
```

Thus, the maximum eigenvalue is close to 2 but not quite (this difference is large enough that it is *not* likely to numerical precision). It is located in the 36th position (index 35) of the eigenvalues 1-dimensional array. To display the actual eigenvector centralities, we can now look at the 36th column of

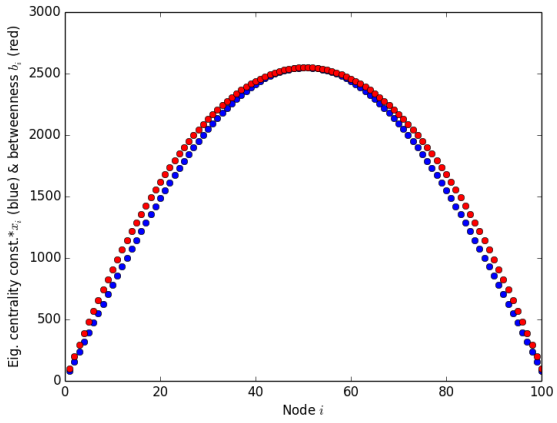


Figure 10: Comparison of centralities for the $n$-line network. The eigenvector centralities scaled by a factor $f = 18116.2393\ldots$ (displayed in blue) grow a bit more sharply than the betweenness centralities (shown in red).

the matrix. But instead of looking at the raw numbers, it is better to plot them. We do this with the following code

```
>>> plt.plot(range(1,101),list(Rh[1][:,imax]),'o')
>>> plt.show()
```

This produces the plot in Fig. 9. The results are very similar qualitatively to the betweenness for the $n$-line so it is natural to ask whether they are indeed the same. To tackle the question, we must first note the differences in scales. The maximum eigenvector centrality has a value of $(x_i)_{\max} = 0.1407\ldots$ and the maximum betweenness centrality is $(b_i)_{\max} = 2549$. Therefore, in order to devise a method of comparison, we calculate the ratio $f = (b_i)_{\max}/(x_i)_{\max} = 18116.2393\ldots$ and proceed to multiply all $x_i$ by $f$. Now, plotting $f \times x_i$ and $b_i$ together in Fig. 10, we can see a direct comparison of the two centralities. Eigenvector centralities display a slightly faster growth than betweenness. At this point, we cannot elaborate on this difference any further. On the other hand, this analysis does show that there is general agreement between these two ways of measuring node importance. The code used to generate this was as follows

```
>>> ximax=0
>>> for xi in list(Rh[1][:,imax]):
...     if xi>ximax:
...         ximax=xi
```

```
...
>>> bimax=0
>>> for i in B.keys():
...     if B[i]>bimax:
...         bimax=B[i]
...
>>> f=bimax/ximax
>>> fxi=[]
>>> for xi in list(Rh[1][:,imax]):
...     fxi.append(f*xi)
...
>>>  plt.ylabel('Eig. centrality const.*$x_i$ (blue) &
    betweenness $b_i$ (red)')
>>> plt.xlabel('Node $i$')
>>> plt.plot(range(1,101),fxi,'o')
>>> plt.plot(B.keys(),B.values(),'or')
>>> plt.show()
```