EDGE BETWEENNESS CENTRALITY AND ITS APPLICATIONS TO COMMUNITY DETECTION IN LARGE NETWORKS

JAVIER CHICO VÁZQUEZ *

Abstract. This paper presents a review of the centrality algorithm, and its to community detection on networks. The algorithm is then tested on graphs generated by the stochastic block model, and compared to other competing procedures for graph partitioning. Some speed up procedures are proposed.

Key words. Network Science, Numerical Methods, Centrality, Community detection

AMS subject classifications. 05C82, 91D30

1. Introduction. In this paper we will introduce and review the Centrality Algorithm with a special focus on its applications to Community detection in networks as discussed by Girvan and Newman [10]. The intuitive idea behind this algorithm is to find the most *relevant* nodes or edges in a network.

The concept of community detection is intuitive, and for graphs small enough to be visualized we could even split them into communities in a fairly accurate manner just by inspection. An example of this is Figure 1, where we can find the communities just by looking at the edge density in different regions of the network. Unfortunately translating this intuition into mathematics is no trivial task.

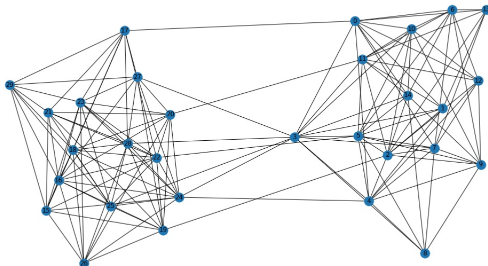

Fig. 1. *A network with two communities*

The benefits of having a fast and accurate community detection procedure are self evident, for a vast array of 21st century applications will benefit from it [14, 4]. Community detection is commonly used to study and analyze biological, social and electrical networks [10]. It plays an essential role in understanding the spread of on networks, for example power grid failures and blackouts [1]. These networks are often large, and thus a fast algorithm is required for a detailed analysis.

Moving on to the centrality algorithm, it has both a node and an edge version. We will focus on the edge algorithm, as it has a more direct application to community detection. The qualitative reasoning behind this is that if we are given an oracle (or black box) that can find the most "important" link in a graph, we can remove that edge, and repeat the process until the graph is disconnected, hence splitting the graph in a meaningful way. [10]

*Department of Mathematics, Massachusetts Institute of Technology/Imperial College London (jchico@mit.edu).

Before going into the specifics of centrality it is important to note that it might be tempting to state that important nodes will have high degree (a large number of neighbors) and dismiss this problem as trivial. However there is a bounty of real networks that contradict this claim. Perhaps the best example is the network of internet cables that connect the world's routers: a node connected to one of the ~ 30 underwater internet cables that crosses the Atlantic might not necessarily have many neighbors, but a lot of "information" will flow through it. The idea behind centrality is to capture that importance which might not necessarily be accurately portrayed by a node's degree. However, this is not the only way to find communities in a graph. With this in mind, two competing community detection algorithms will be presented as alternatives: modularity maximization and Laplacian graph partitioning.

Several variants of the centrality algorithm have been presented in the decades since it was first introduced [3], as well as a generalization to weighted networks. We will comment on some of them at the end of the review, paying particular interested to those with stochastic components like [16].

This paper is organized as follows. Essential network science concepts are revised in section 2. The main algorithm is explained and implemented in section 3. The algorithm's community finding ability is tested in section 4. A discussion of a competing algorithms is given in section 5 followed by a brief conclusion in section 6.

2. Network Science Preliminaries. Suppose we have a network with N nodes and L edges (links between the nodes), for example in Figure 1. The following are standard definitions - available for instance in [1, 14] - used to describe a network in the language of linear algebra.

Definition 2.1 (Adjacency Matrix). *We define the adjacency matrix A_{ij} of a graph which for the time being we assume to be unweighted (all edges are equally important) and undirected (all edges can be travelled in both directions) in the following way: [1].*

$$A_{ij} = \begin{cases} 1 & \text{there is a link between nodes } i, j \\ 0 & \text{otherwise} \end{cases}$$

Definition 2.2 (Degree of a node). *For a node i, we define the degree of node i denoted as k_i by the number of neighbors it has. For an unweighted and undirected network we can express this in terms of the adjacency matrix as*

$$k_i = \sum_{j=1}^{N} A_{ij}$$

We further define the degree matrix, $D = \operatorname{diag}(k_1, \ldots, k_N)$ where k_i is the degree of node i. Finally, we can use this definition to obtain the Laplacian matrix:

$$L = D - A$$

This is an $N \times N$ matrix, where the diagonal entries will be k_i (the number of neighbours node i has), and the off-diagonal entries will be -1 if there is a link between node i and j, and zero otherwise. [1]

Definition 2.3 (Connected Network). *A graph is connected if and only if there exists a path joining any two nodes in the graph [14].*

3. Edge Betweenness Centrality. As discussed in the introduction, the main idea behind centrality is to find the most important "connectors" in a graph. We can make this more precise by describing important edges as those by which many shortest paths joining other edges go through. As previously explained these nodes might sometimes have a low degree, so finding it just by looking at high degree nodes is not acceptable. For this, we turn to *betweenness centrality* (BC). We define the *edge betweenness centrality* of an edge that connects two nodes α and β as [12, 14, 1, 10]

$$y_{\alpha\beta} = \sum_{a=1}^{N} \sum_{b=1}^{N} \frac{m_{\alpha\beta}(a, b)}{p_{ab}} \tag{3.1}$$

where $m_{\alpha\beta}(a, b)$ is the number of shortest paths between nodes a and b that go through the edge (α, β), and p_{ab} is the total number of shortest paths between nodes a and b. Hence, we are effectively counting the fraction of shortest paths between two nodes that go through a particular link, and then summing over all possible pairs of nodes.

Once we have made the definition precise, we can think about how to compute it. The straightforward way would be by considering every possible pair of nodes (there are $\binom{n}{2}$ possible pairs of nodes), and computing every single shortest path between them. Then we focus on each of the L edges and for each pair of nodes compute the share of shortest paths that go through that particular edge. This is conceptually simple, but inefficient and expensive [14]. Fortunately, there are efficient algorithms to accelerate the computation of betweenness centrality, as discussed in subsection 3.2. However we will first go over how to apply centrality to detect communities.

3.1. Edge Betweenness Centrality as a Community detector. As outlined in the introduction, we can detect communities by successively removing nodes edges with high centrality. Once the edge betweenness centrality has been computed for every edge, the network can be divided by removing the edge with the highest betweenness centrality [10]. If the new network is still connected, the betwenness centrality of the edges in the new network is computed, and a new edge is removing using the same criteria, and this process is repeated until the resulting network is disconnected. By then, it has been divided into two communities. Assuming we have an efficient function to compute the edge betweenness centrality for a network, denoted by edge_bc, the pseudocode for splitting it into two communities is detailed in Algorithm 3.1.

Algorithm 3.1 Finding 2 communities

```
Start with G, define H = copy(G)
while G is connected do
  Compute bc = edge_bc(H)
  Choose e* = argmin_{e∈edges(H)}(bc)
  Update H = H.remove_edge(e*)
end while
return H
```

If more than two communities are necessary, say r of them, this process can be continued until the output network H has r disconnected components [10].

3.2. Efficient recursive computation of Edge Betweenness Centrality. It has now become clear that a fast way to compute the edge betweenness centrality is desired to split networks cheaply and efficiently. With this aim in mind, we will define

the source edge betweenness centrality. For this, we choose a source node a, and use a recursive procedure to compete the source edge betweenness centrality efficiently, with the algorithm detailed in this section. Once this is done we can choose a another source node and iterate. We define the source edge betweenness centrality by [10]

$$y_{\alpha\beta a} = \sum_{b=1}^{N} \frac{m_{\alpha\beta}(a,b)}{p_{ab}}$$

The intuition behind this definition is just fixing a in (3.1). The key idea to compute the source edge betweennes centrality is to build a directed shortest path graph (DSPG) which encodes the shortest paths from each node to the source node a. Then starting from the leaf nodes (in the directed graph these are the nodes which have no edges *into* them), we can compute the source edge betwenness centrality for an edge joining nodes α and β recursively in the following way [10]:

$$y_{\alpha\beta a} = \sum_{b=1}^{N} \frac{m_{\alpha\beta}(a,b)}{p_{ab}} = \frac{p_{a\alpha}}{p_{\alpha\beta}} \left(1 + \sum_{b \in \mathbb{N}_\beta} y_{\alpha\beta b} \right) \tag{3.2}$$

Where $\mathbb{N}_\beta$ is the set of neighbours of β in the directed shortest path graph.

Computationally we don't assemble the directed shortest path graph. Instead we use a two stage procedure. First, we do a forward sweep from node a, (known as breadth search first (BSF)), to compute

1. Which nodes are reachable from the source node. This is stored in a list L_1. Do note that in our testing it is assumed the graph is connected and hence L_1 should be a list of ones.
2. p_{ab} for all b, the number of shortest paths between nodes a and b. These variables are stored in a list L_2
3. Store the order in which nodes are added to the queue (the order in which they are explored). This ordering is stored in a list M.
4. Store the distances from the explored nodes to the source node a in a list L_3.

Each of these variables are stored in arrays or lists. The pseudocode for the forward sweep is available in Algorithm 3.2. It is important to note that this forward sweep is intimately related to Dijkstra's algorithm [5] to find the shortest paths between two nodes in a graph, although it works in a more general way and its output is optimized to be used in the second stage of source edge betwenness centrality computation, which is discussed next. Moreover, the specific shortest paths between any two nodes is not relevant for centrality calculation.

The second stage is a backward sweep starting from the leaf nodes (these are the nodes were you can't travel any further away from the source node once you reach them). For each edge (α, β) connected to a leaf node β, set the edge centrality to [10]

$$y_{\alpha\beta a} = \frac{p_{a\alpha}}{p_{\alpha\beta}}$$

Then for nodes β closer to the source, identify the neighbours $b \in \mathbb{N}_\beta$ and add $y_{\beta ba}$ to a running total. For each α where $\beta \in \mathbb{N}_\alpha$, add 1 to the final and multiply by $\frac{p_{a\alpha}}{p_{\alpha\beta}}$, using (3.2). The pseudocode for the backward sweep is available in Algorithm 3.3 .

Finally, we can get the edge betwenness centrality by iterating over all nodes a as source nodes and computing (a one half factor is included to account for over

Algorithm 3.2 Forward sweep: starting from node a, graph G with N nodes.

```
Initialize an array L_1 = zeros(N) marking all nodes as unsearched
Initialize an array L_2 = -1 storing the distance from the source node
Initialize an array L_3 = zeros(N) storing the number of shortest paths
Initialize a list M = [a] storing the order in which nodes are searched
L_1(a) = 1 as the source node is reachable from itself
L_2(a) = 0
L_3(a) = 1
Set i = 1
while i < length(M) do
  n = M[i]
  i = i+1
  for j with A_jn = 1 do
    if L_1(j) = 0 then
      L_1(j) = 0
      L_2(j) = L_2(n) + 1
      L_3(j) = L_3(n)
    else if L_2(j) = L_2(n) + 1 then
      L3(j) = L_3(j) + L_3(n)
    end if
  end for
end while
return L_1, L_2, L_3, M
```

counting)

$$y_{\alpha\beta} = \frac{1}{2}\sum_{a=1} y_{a\alpha\beta}$$

3.3. Floating point operation count and cost analysis. The performance of the algorithm will be evaluated by counting the total number of floating point operations, as using timing measurements is not reliable for code which is not highly optimised (as is the case now). The overall cost to compute the edge betwenness centrality of a graph with N nodes and L edges using the algorithm described in subsection 3.2 is $\mathcal{O}(N(N+L))$.

We can see this by noting that the computation requires using the N nodes as source nodes, and each source edge betwenness centrality computation requires $\mathcal{O}(N+L)$ flops. Hence, $\mathcal{O}(N(N+L))$ in total to compute edge betweenness centrality.

It is important to node that for most networks $N < L$ (and very often $N << L$) so the dominant factor is NL. Indeed if the random graph is generated randomly using the most basic random graph generation scheme, Erdos-Renyi [6], then $L \sim p\binom{N}{2} \sim N^2$ and the cost to compute the edge betwenness centrality scales as as $\mathcal{O}(N^3)$. Here $p \in [0,1]$ is the probability of placing an edge between any two nodes.

It is also important to see why the computation of the source edge betwenness centrality is $\mathcal{O}(N+L)$. We can justify this by noting that for a complete graph, the forward sweep requires us to visit the N nodes in the graph, and for each node we must consider all its neighbours. Hence, if we assume the average degree of a graph is $\mathcal{O}(1)$ (reasonable for large real networks), this will require $\mathcal{O}(1)$ flops per node, and $\mathcal{O}(N)$ in total for the forward sweep.

Algorithm 3.3 Backward Sweep, using as inputs the outputs from Algorithm 3.2

```
edge_bc = zeros(L)
for j = 1, ..., N do
   β = M.pop(end)
   running_total = 0
   alphas = []
   for b with A_{bβ} = 1 do
      if L_2(b) = L_2(β) + 1 then
         edge = (β, b)
         running_total = running_total + edge_bc(edge)
      else if L_2(b) = L_2(β) − 1 then
         alphas.append(b)
      end if
   end for
   for α ∈ alphas do
      edge = (α, β)
      edge_bc(edge) = L_3(α)/L_3(β) (1 + running_total)
   end for
end for
return edge_bc
```

Similarly, the backward sweep requires us to essentially consider each edge in the graph, and thus it requires $\mathcal{O}(L)$ operations in total. Hence, $\mathcal{O}(N+L)$ flops for each source edge betwenness centrality calculation.

For the task of community detection we would need to repeat the calculation K times, where K is the number of edges we remove before the graph becomes disconnected. In general we can't know K before hand, but we expect it to be much smaller than either N or L.

Several speed up procedures come to mind to reduce this cost. For community detection, we might suggest not repeating the computation after removing one edge, and instead also removing the edge with the second largest betwenness centrality from the graph, and only recalculating centrality every other, or every three edges removals. This will definitely reduce the cost but the effect it will have when detecting relevant communities is hard to predict.

To reduce the cost of the edge betwenness centrality computation itself, a possible suggestion is to not use nodes with low degree as source nodes, or perhaps use only a subset of the nodes as source nodes, sampled randomly for each calculation. Again, this will certainly reduce the cost but the centrality calculation will be incorrect. However, this might not be as catastrophic as it might appear, since for the community detection task only the relative values of the centrality are required. In particular we only care about the edge with the largest centrality. Hence we might hope that using only a sample of nodes as source nodes might still give us the node with the highest centrality accurately.

3.4. Implementation. The source edge betwenneess centrality algorithm was implement in `Python`. It is built upon the `Python` module `Networkx`, a package designed to work with large networks. The implementation only uses the package to access the graph, as the data structures offered are easy to work with. The code is available in the corresponding .zip file.

4. Evaluating the performance of the algorithm for the community detection task. The algorithm's ability to detect communities within a graph will be tested using graphs generated by the stochastic block model (SBM).

4.1. The stochastic Block Model. The stochastic block model was first introduced by Holland et. al. [11]. It is a model for graph generation which by construction creates structured graphs, meaning that it is more likely to place edges within communities than between them. Although there are simpler models to generate random graphs, such as Erdos-Renyi [6], Barabasi-Albert [15], or Watts-Strogatz [19] which can account for phenomena observed in real world networks such as high clustering, "small worlds" and power-law degree distributions, they are not prone to generating clear communities, nor do they incorporate a parameter or mechanism that controls the formation of communities. Hence, it is necessary to use the SBM to test the centrality algorithm.

Mathematically the SBM takes the following parameters as inputs:

- The number of communities, r
- The number of nodes within each community, N_i for $i = 1, 2, \ldots, r$
- P, a symmetric $r \times r$ matrix, such that its (i, j) entry P_{ij} is the probability of having an edge between nodes in communities i and j. In particular, for this model to produce networks with a clear community structure we expect P to be diagonally dominant, so that edges are more likely to be placed within a community instead than between them.

We then generate a random sample from the above distributions. For the purpose of testing the algorithm we will use a simple form for the matrix P:

$$P = pI_r + q(J - I_r)$$

where J is a matrix of ones, i.e. $J_{ij} = 1$ for all its entries. p is the probability of placing an edge within a community, and q the probability of placing an edge between communities. In particular, during testing r will be constrained to $r = 2$. We can denote this parameter space by (r, p, q). We can see examples of graphs generated by this setup in Figure 1 and Figure 2

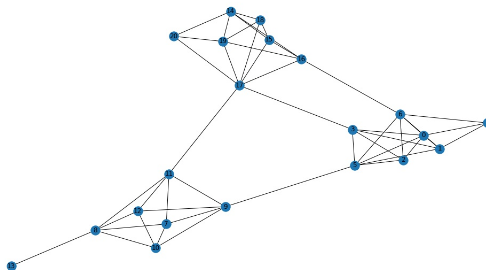

FIG. 2. *Network generated using the stochastic block model with 3 communities*

4.2. Testing methodology. To test the centrality algorithm for a network with 2 communities ($r = 2$) the following scheme was used:

- Fix a number of nodes per community $N_i = N$, independent of i for now.
- Fix a value for p, the probability of placing an edge between nodes in the same community.

- Select the values of q we are interested in analyzing ,$\{q_i\}_i$.
- Fix M, the number of SBM graphs generated for each value of q
- For each q_i generate M SBM graphs with parameters $N_i, r = 2, p, q_i$. Split the graph into two communities using the edge centrality algorithm from section 3 and check if it matches the communities used as inputs for the model. Store the average success rate for each q_i

It is interesting to add another degree of freedom to the problem. In the above scheme, all communities have the same number of nodes. This is not realistic, i.e. in a school the two most clear communities are teachers and students, which are not equal in number. To account for this we can draw N_i randomly for each i to generate networks with communities of different sizes, and repeat the aforementioned testing procedure with this new stochastic aspect.

4.3. Results. The agglomerated testing results are presented in Figure 3. The algorithm was tested using the aforementioned scheme for several values of the parameters, with particular emphasis placed on varying N_i. The probability of placing an edge between two nodes in the same community, p was fixed to 0.65.

Generally, the accuracy of the communities drops rapidly at around $q \sim 0.20$. For values greater than $q \geq 0.30$ the algorithm looses all predictive power when finding communities.

As a general rule, the algorithm will be more precise when the graphs have more nodes, but this is expected as the communities will be more pronounced and hence there will be relatively fewer "connection" nodes, and thus the centrality algorithm will be more accurate.

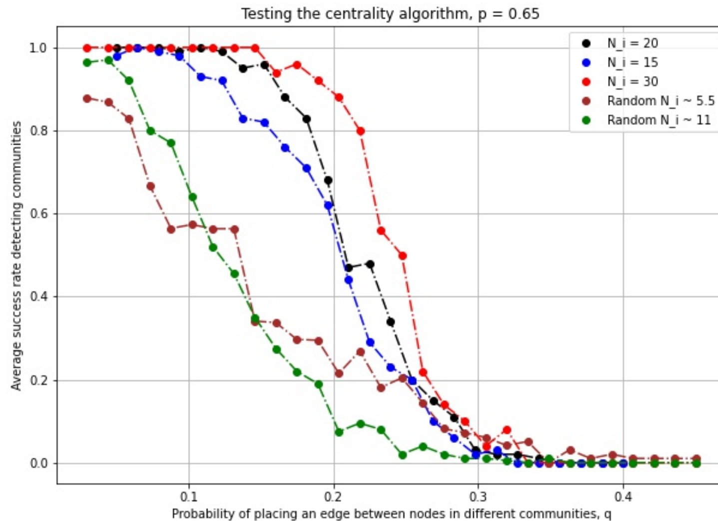

Fig. 3. *Testing results. The average success rate when predicting the communities is plotted against the probability of placing an edge between nodes in different communities*

5. Competing algorithms for Community detection. In this section two algorithms for community detection are presented schematically. Both can be formulated as optimization problems which can be solved efficiently using numerical linear

algebra. Comparisons are made to the centrality algorithm, and the benefits and shortcomings of each procedure are discussed. The algorithms are presented from a theoretical perspective and with a very strong emphasis in community detection.

5.1. Modularity Maximization. We define the modularity of a group of nodes S_a of a graph by [14, 10]

$$M_a = \frac{1}{2L} \sum_{i \in S_a} \sum_{j \in S_a} \left(A_{ij} - \frac{k_i k_j}{2L} \right)$$

If we set all nodes into one group its modularity will be zero, so the idea is to split the graph into two groups S_a and S_b in such a way that the total modularity

$$M = M_a + M_b$$

is maximized. (M is bounded above by one [1]). This is done via Modularity maximization. We will restrict the problem to splitting the graph into 2 communities, a and b. The key idea is building the modularity matrix, B, with [13, 17]

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2L}$$

Now we define the partition vector $\mathbf{s}$, with $s_i = 1$ if node i is in group b, and $s_i = -1$ otherwise. Then $\frac{1}{2}(s_i s_j + 1)$ is 1 if both nodes in the same group, zero otherwise, and the modularity for a given partition is

$$M = \frac{1}{4L} \sum_{i \in S_a} \sum_{j \in S_a} \left(A_{ij} - \frac{k_i k_j}{2L} \right) (s_i s_j + 1) = \frac{1}{4L} s^T B s$$

Which we wish to maximize subject to $s_i = \pm 1$. This is hard to solve in general. One approach relaxes the constraint to maximizing the above subject to $|\tilde{s}|^2 = N$, with the entries of this vector allowed to be real numbers, not just ± 1 (we will correct this later) [13]. Then we can maximize this with the largest eigenvalue of the modularity matrix, λ_1. We re-scale the corresponding unit eigenvector v_1 to $\sqrt{N} v_1$ to satisfy the constraint and then

$$\tilde{M} = \frac{1}{4L} \tilde{s}^T B \tilde{s} = \frac{N \lambda_1}{4L}$$

Finally, we adjust $\tilde{s}$ by choosing $s_i = 1$ if $\tilde{s}_i > 0$ and $s_i = -1$ otherwise. This procedure is known as the *sign-cut* or *zero-threshold cut* [17] .

5.2. Laplacian graph partitioning. As opposed to modularity maximization this will be a minimization problem. The idea is to split a graph into two groups of nodes (a and b, as before) where the number of links crossing from one group to the other (defined as the cut size, c) is minimized. Mathematically, we set $s_i = 1$ if node i is in group a and $s_i = -1$ otherwise, as before. Then, for a given partition (which is just a vector $\mathbf{s}$ of ± 1 entries the cut size is [7]

$$c = \frac{1}{4} \sum_i \sum_j A_{ij} (1 - s_i s_j)$$

this is what we wish to minimize. However, it is useful to know that

$$\sum_i \sum_j A_{ij} = K = \sum_i \sum_j s_i s_j \delta_{ij} k_i$$

Going to back to Definition 2.2, and remembering the Laplacian matrix $L = D - A$, it turns out that we can write the cutsize as [13]

$$c = \frac{1}{4} s^T L s$$

In contrast to modularity, we want to minimize this. The trivial solution is all nodes in a single group, where $c = 0$, which works by taking $s = \frac{1}{\sqrt{N}}\text{ones}(N) \in \ker(L)$. However if we don't allow this, and the problem is constrained to positive cutsizes, we can use use $\tilde{s} = V_{N-1}$, the eigenvector associated with the second smallest eigenvalue (known as the *Fiedler vector* [2]), λ_{N-1}, known in the literature as the *algebraic connectivity* [7], normalized to have length $\sqrt{N}$. Then the cut size is $c = \frac{1}{4} N \lambda_{N-1}$. Finally, we can repeat the final step in modularity maximization to get a vector of ± 1's entries, using the sign-cut approach.

5.3. Cost for each competing algorithm. As seen in section 3 the cost to compute edge betwenness centrality is $\mathcal{O}(N(N + L))$, and the cost to divide a graph using edge betwenness centrality is $\mathcal{O}(KN(N + L))$, where K is the number of edges we remove before the graph becomes disconnected. When a graph has community structure we expect K to be small compared to both N and L.

For modularity maximization the problem reduces to the power method. Hence, for each iteration we will have to perform a matrix-vector multiplication, requiring $\mathcal{O}(N^2)$ flops in general. However for sparse networks, as in the case of most real networks, this can be reduced to $\mathcal{O}(N + L)$. Finally, if we require q iterations of the power method until the convergence is good enough, the total cost for performing modularity maximization will be $\mathcal{O}(q(N + L))$. However, on networks with the small-world property[1] it has been observed that $q \sim \log N$, giving a final floating point operations estimate of $\mathcal{O}(\log N(N + L))$ [17]

For the last algorithm considered, without going into too much detail as it is not the main topic of this paper, and restricting the problem to the case $\lambda_{N-1}(L) < 1$ (very general), we can then transform the problem into a power iteration like scheme with similar cost as modularity maximization. [8]

5.4. Differences between the algorithms. In this brief section we will walk through some of the main differences between the three algorithms discussed. Particular attention will be paid to the shortcomings and problems for each of the algorithms, as this will be helpful to identify cases where using the centrality algorithm will be recommended.

Starting with the modularity algorithm, an advantage is that it can be modified to the more interesting case of a weighted graph without much theoretical complication, where instead of an adjacency matrix A we have a matrix with weights W (which we can assume has non-negative entries), then the modularity of a group of nodes is

$$M_a = \frac{1}{2L} \sum_{i \in S_a} \sum_{j \in S_a} \left(W_{ij} - \frac{k_i k_j}{2L} \right)$$

where $k_i = D_{ii} = \sum_{j=1}^{N} w_{ij}$, and perform modularity maximization on this matrix. The same is true for Laplacian graph partitioning using a weighted Laplacian.

[1]In essence, a network has the small world property when the length of the largest shortest path in the graph scales as $\sim \log N$, where N is the number of nodes.

Both Modularity Maximization and Laplacian Graph partitioning benefit greatly from being expressed as linear algebra problems (over $\mathbb{R}$), as it means the widely developed and used tools from numerical linear algebra are applicable. Hence, as seen in the previous section they are both less expensive to run in most networks. However they have important flaws that must be considered before they are used in real problems.

Modularity maximization suffers from a resolution limit which greatly reduces its ability to detect small communities in large networks. Essentially, small communities are lumped together into one big community if the graph is sufficintly large. This shortcoming is explored in detail by Fortunato et. al. [9]. In practice this means that when we use modularity to find communities in a large graph we will be unable to "resolve" small communities, even if we expect them to exist.

As for Laplacian graph partitioning, it has several important shortcomings. Most importantly, placing a node of degree one (just a single neighbour) on a group by itself achieves the minimum cutsize, exactly one. (And nodes of degree one are common in most real networks). Evidently this is not desirable and a possible fix is to place all nodes with $\tilde{s}_i < \text{Median}(\tilde{s})$ in a single group, instead of the standard sign-cut approach. However, this fix will create communities of equal sizes, which is not desirable either as it might not be representative of the community structure. In light of this, it is reasonable to claim Laplacian graph partitioning suffers from important pitfalls which heavily restrict its ability to compete both with modularity and centrality based methods when detecting communities in real graphs.

6. Conclusions and future work. All in all we have compared the edge betweenness centrality approach to community detection with two numerical linear algebra based partitioning schemes. We can summarise the finding by saying that edge betweenness centrality is a robust procedure for community detection, although it is more expensive to run in comparison to the other algorithms, which in turn suffer from more fundamental limitations and problems.

In the years since the algorithm presented here was introduced many advances in the field of network science have taken place, driven by the mayor role data has played since the start of the information age. In particular, it is good to highlight the addition of stochastic methods to accelerate centrality computations. Newman introduced a new algorithm which uses random walks to estimate the node centrality of a network [16]

When should you use the centrality algorithm. For networks with a geographical context (internet cables, some social and biological networks), centrality will provide insight by finding the most important nodes. As mentioned before, the underwater transatlantic internet cables are perhaps the best example. In social or biological networks we can think of edges with high centrality as choke points: they might represent a mountain pass, a river crossing, a border point between two countries... Hence it is reasonable to argue centrality better fits this kinds of problems, where the communities might be defined by geography, instead of using an optimization based a approach which might be blind to this aspects.

Future work. As outlined in subsection 3.3, it would be interesting to study what is the effect of only using a fraction of the nodes as source nodes in the algorithm. This has the potential to accelerate the computation drastically. Furthermore, a second possible interesting approach is to not repeat the computation of the edge betweenness centrality after removing one edge from the graph, and only repeat the computation after 4,5, ... edges have been removed (or some other multiple). This could also

be incredibly beneficial for speeding up the community detection procedure. However due to time and computation power constraints studying these ideas exhaustively was not possible. Some related work was done by Ulrik et. al. [3].

Another avenue of further study is incorporating stochastic methods as presented by Newman [16], where he proposed using random walks to measure centrality. This idea is both intuitive (we expect important nodes to be visited more often by random walkers) and elegant, for it benefits from the recent advances such as Ergodic Theory and Markov Chains [18].

Finally, studying the stability of centrality measures to perturbations in the weights of edges, the number of nodes or edges is also important. This is because most real networks are built from datasets which might be incomplete or contain mistakes, and before drawing conclusions from centrality applications it is essential to understand how the missing information might affect the result.

Acknowledgments. The author is grateful to Dr. Andrew J. Horning for his support and guidance in this project.

REFERENCES

[1] A.-L. Barabási, *Network science*, Cambridge University Press, Cambridge, United Kingdom, 2016 - 2016.

[2] A. Bertrand and M. Moonen, *Distributed computation of the fiedler vector with application to topology inference in ad hoc networks*, Signal processing, 93 (2013), pp. 1106–1117.

[3] U. Brandes, *On variants of shortest-path betweenness centrality and their generic computation*, Social networks, 30 (2008), pp. 136–145.

[4] A. Clauset, M. Newman, and C. Moore, *Finding community structure in very large networks*, Physical review. E, 70 (2004), pp. 066111–066111.

[5] E. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik, 1 (1959), pp. 269–271.

[6] P. Erdös and A. Rényi, *On random graphs i*, Publicationes Mathematicae Debrecen, 6 (1959), p. 290.

[7] M. Fiedler, *Algebraic connectivity of graphs*, Czechoslovak mathematical journal, 23 (1973), pp. 298–305.

[8] M. Fiedler, *Laplacian of graphs and algebraic connectivity*, Banach Center publications, 25 (1989), pp. 57–70.

[9] S. Fortunato and M. Barthelemy, *Resolution limit in community detection*, Proceedings of the National Academy of Sciences - PNAS, 104 (2007), pp. 36–41.

[10] M. Girvan and M. Newman, *Community structure in social and biological networks*, Proceedings of the National Academy of Sciences - PNAS, 99 (2002), pp. 7821–7826.

[11] P. W. Holland, K. B. Laskey, and S. Leinhardt, *Stochastic blockmodels: First steps*, Social networks, 5 (1983), pp. 109–137.

[12] M. Newman, *Fast algorithm for detecting community structure in networks*, Physical review. E, 69 (2004), pp. 066133–066133.

[13] M. E. J. Newman, *Finding community structure in networks using the eigenvectors of matrices*, Physical review. E, 74 (2006), pp. 036104–036104.

[14] M. E. J. Newman, *Networks*, Oxford University Press, Oxford, United Kingdom ;, second edition. ed., 2018.

[15] M. E. J. M. E. J. Newman, *The structure and dynamics of networks*, Princeton studies in complexity, Princeton University Press, Princeton, N.J, 2006.

[16] M. J. Newman, *A measure of betweenness centrality based on random walks*, Social networks, 27 (2005), pp. 39–54.

[17] A. POTHEN, H. D. SIMON, and K.-P. LIOU, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM journal on matrix analysis and applications, 11 (1990), pp. 430–452.

[18] X. Wang, G. Chen, and H. Lu, *A very fast algorithm for detecting community structures in complex networks*, Physica A, 384 (2007), pp. 667–674.

[19] D. J. Watts and S. H. Strogatz, *Collective dynamics of 'small-world' networks*, Nature (London), 393 (1998), pp. 440–442.