# Community Detection for Social Media Networks

Seth Singson-Robbins
Fordham University
ssingsonrobbins@fordham.edu

**Abstract – The paper looks at how community detection for graph algorithms is implemented and the use cases of this methodology for social media and marketing companies. An explanation of vertices, edges, and graphs implemented in Python are introduced and a breakdown of several algorithms to score network edges to rank the edges based on their connection power and likelihood to be connecting unrelated communities together. It then investigates the edge removal process to allow for these communities to be separated and identified. Use cases of the algorithm include pulling similar users from the network, or look alike audiences, as a tool to synthesize these communities into new potential customers for advertisers.**

## I. Introduction

Community detection is a way to determine how communities are connected based on the relationship between the vertices via the edges within the network. These communities are densely interconnected and could be part of either smaller or larger communities. Being able to identify these communities can have many use cases depending on the dataset. One of the benefits of this type of network analysis is that it is unsupervised, meaning that prior knowledge about the network is not necessary and relies purely on the edges and connections with the network to make these connections.

For social media networks, graphs are compiled with vertices being the users of the network and the edges showing interactions, in a lot of cases friendships, between the users. These relationships can be very indicative of being part of a larger community. Community detection can help remove the weak links between these users uncovering the true communities and connections that make up a social media network. There are a lot of use cases for identifying these communities. Social media networks can help identify users in specific communities and sell them as part of an audience segment. They can also identify users in specific user groups to figure out how to make specialized products that would fit the needs of these users while they use the site. For marketers and advertisers, they can buy these audience segment to sell their products that fit this group. They can also utilize the list of users that they already identified being on their own site (or previous purchasers of their products) helping them find similar users through what is called 'look alike audiences', untapped users in the same community leading to new potential customers. This relationship is a win-win for the user, who gets a specialized experience on the social network from special features on the site to user curated advertising for products they would be interested in buying leading to a surplus to the user experience on the social network. Social media networks get to sell these groups as product packages while simultaneously improving their site with features their users want while marketers get to identify their target audience increasing their potential customer base more easily.

## II. Building Graphs

To understand how community detection will be calculated, some knowledge on the production and features of our graphs must be explained. The graph data is based on three Python objects with functionality and attributes that will help with the analysis of the graph and identification of the communities.

### A. Vertex Class

The vertex class is for each vertex or user in the network with its name being included in the value attribute. It has a list of its neighbor vertices, or vertices that are connected to the object via an edge, along with a list of edges it is connected to. Several attributes were included for the community detection algorithms including the name of the community, which is 0 when the entire graph is connected, a status explaining if it was visited which is used for search-based functions, and a score which is used as a component for scoring the connectivity of an edge in the same graph. It also has the option of including audiences for further analysis of the communities.

In terms of functions, there are several functions that help with the organization of the attributes including adding and removing neighbors once edges are adding to the network and checking if a vertex or an edge is a neighbor. The function addAudience was also implemented to help with additional analysis of the final communities that were established.

### B. Edge Class

The edges class is for each edge or connection between the vertices in the graph. Its main attribute is the two vertices it connects. For community detection, it also includes a status if visited for search-based functions and several score attributes which will be explained further in the scoring edges section.

There are also several important functions the edge object must run for community detection.

- getVertices lists the two vertices the edge is connected to.
- edgeNeighbor shows all the edges that are connected to the two vertices the edge is connected to.
- secondVertex pulls the 2nd vertex based on the vertex provided so it can determine the other vertex that the edge connects to.

### C. Graph Class

The graph class combines all the vertices and edge objects into one larger network. Its attributes include its list of vertices and edges. It has a significant list of functions that look to add new vertices and edges to the graph and functions to pull the list of edges and a list of edge neighbors of a vertex that is in the graph.

## III. Scoring Edges

When determining the new communities, each of the edges needs to be scored. These scores determine how important the edge is in connecting users together. The higher the score, the more likely that this edge is an important link and, if removed, will result in users being disconnected from each other. What it also means is that these users are not as densely connected with each other thus would less likely be part of the same community. Cutting the connections also brings insight into which are the actual communities since, once enough edges are connected, the true communities will become completely separated from each other. To get the scores for each edge, four algorithms much be created and implemented on the graph which will be discussed in more detail: breadth-first search, creating a score for the vertices, creating a score for the edges, then summing up the edge scores for final cumulative scores of each edge.

The model itself is built based on the Girvan-Newman Algorithm and Edge Betweenness Centrality written by Analytics Vidhya.[1]

## A. Breadth First Search

Breadth First Search (BFS) is an algorithm that searches through the graph to determine the shortest path from an initial vertex to all the other vertices in the graph that can be traveled to from the graph edges. The algorithm starts with the initial vertex set by the user and finds all the vertices that are connected to the vertex by an edge and saving the distance (which will in this case be one or it needing one edge to get to from the initial vertex to this new vertex). It then sets the vertices to 'visited' so that it knows that it has already discovered this vertex ensuring that, if it is a neighbor to another vertex in its path that the algorithm knows that it was already there, it does not connect the vertices or redetermine the path length. Every step thereafter, the newly visited vertices then looks for all its neighbors and does the same thing noting the distance from the initial vertex and that it was visited. This step keeps iterating until all the vertices that have paths are found. The algorithm returns an array with each item being a list of vertices found at that step signifying the distance from the initial vertex (so item 0 is the vertices found at the first round of searching).

One difference between this model's algorithm and the typical BFS algorithm is that the community number is saved when a vertex is visited. This will be utilized in the community count and community listing which will be described further in section IV. This algorithm goes through every vertex and checks for every edge making the algorithm have a run time of $O(|V| + |E|)$.

## B. Vertex Score

The next step is an algorithm to calculate the score of a vertex. The score is symbolic of the number of paths available from the initial vertex to the vertex the model is finding a path for. It does this for every vertex in the graph that the initial vertex can reach. The algorithm itself is very similar to BFS and utilizes its output list to help with its traversal through the BFS tree. At the initial step, the initial vertex is set to 0 and each of its neighbors receive a score of one and set to 'visited'. For every step after then, the newly visited vertices will iterate to its non-visited neighbors and those vertices will be scored based on the summation of all the previously visited nodes to check if they are connected. The score of the vertex will be the sum of the nodes' parents' scores. This continues until every vertex that could be visited is visited. Knowing which is newly visited is from the list that was returned from the BFS step. Along with the vertices being scored, it also returns the last vertex that was visited which will be used in the next edge score section. Like BFS, it has a run time of $O(|V|+|E|)$.

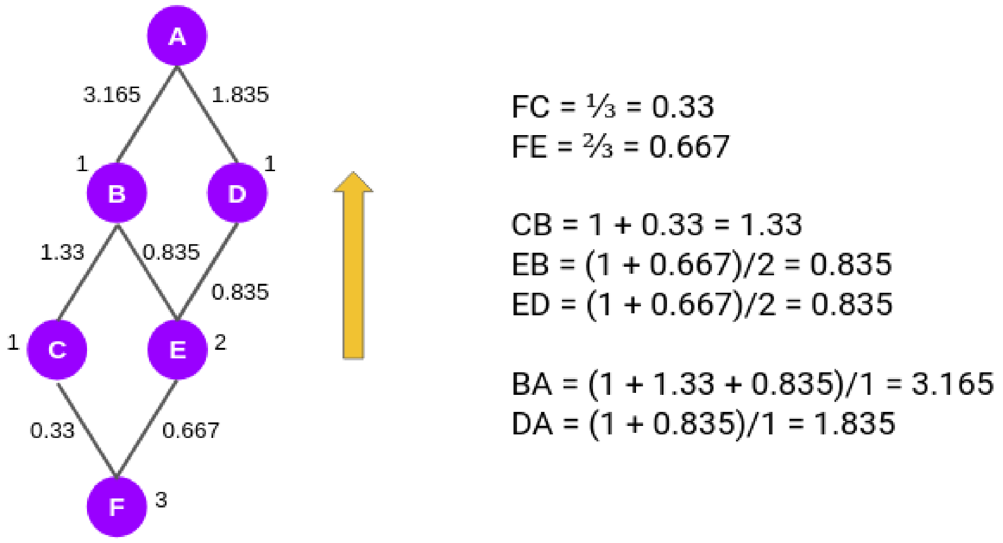
## C. Edge Score

With these previous two algorithms in mind, the next step is to score the edges based on the initial vertex. After running BFS and the vertex score algorithms, the algorithm starts at the last vertex that was returned by the vertex score algorithm. For each of its neighbors, it gives the edge that connects the two vertices a score equal to the neighbor's score divided by the root vertex. For each iteration after, it will search for the next unvisited neighbors comparing the edges that connect them and giving them a score based on the following formula:

$$\text{score} = (1 + ES) / N$$

- N is the number of unvisited edges the parent edges have. If the vertex has two unvisited neighbors, N is set to 2.
- ES is the value of the parent edge. However, if the unvisited edge has multiple parent edges, then it is the sum of those edges' scores.

The image below from Analytics Vidhya shows how edges would be scored based on a BFS tree including what the vertex scores would look like for this graph.


A
3.165
1.835
1
B
D
1
1.33
0.835
0.835
1
C
E
2
0.33
0.667
F
3
FC = ⅓ = 0.33
FE = ⅔ = 0.667
CB = 1 + 0.33 = 1.33
EB = (1 + 0.667)/2 = 0.835
ED = (1 + 0.667)/2 = 0.835
BA = (1 + 1.33 + 0.835)/1 = 3.165
DA = (1 + 0.835)/1 = 1.835

In terms of the algorithm's run time, it must run through all the edges every time it checks the neighbors of a vertex (which since BFS already ran it runs through each vertex once) resulting in a run time of O(|V||E|).

### D. All Edges Score

The final algorithm for edge scoring is to run the edge score algorithm for every vertex in the graph where each vertex is the starting vertex. This gives the chance for every vertex to be the starting point and scoring the edges based on the paths from the initial vertex showing how the vertex is deeply connected or not connected to other vertices within the graph. The final output is summation of the edge score from every round which will be used for analysis and creating communities. Since it runs the edge score algorithm, which has a run time of O(|V||E|), for every vertex in the graph, the run time is O($|V|^2$|E|).

## IV. Creating Communities

Once the edge scores have been calculated, edges can start being removed to cut up the graph and show which communities are more connected. The rational is to remove edges with the highest score since they are the most likely to disconnect less densely connected users that are less likely to be part of the same community. A few algorithms were used to determine the new communities including a community count algorithm, a community divider algorithm, and two algorithms reporting algorithm size and a list of vertexes within each community. Algorithms are utilized assuming the final number of communities is known.

### A. Community Count

The community count algorithm is utilized to confirm the number of fully separated communities the graph currently has. It starts by running a BFS algorithm on the first vertex in the graph's vertices list and seeing what that vertex is connected to. The idea is that a fully separated community is one that a vertex can only reach other members of its community through its edges and not other within the graph network. This is essentially what BFS does when it runs. Once BFS is completed, it will check every vertex in the graph checking if it is has been visited. It also has a counter that increases everytime BFS is completed so it knows what community it is on and how many times it has run BFS. If it has not been visited, it will do BFS on that vertex also finding its connected vertices which make up its community. This way it will do a BFS on every community until it finds all the communities. When it runs BFS, it also saves the community number that it is on within the vertex's object as an attribute, something that will be used later for reporting. The final output is an integer that returns the number of fully disconnected communities within the graph. It runs through every vertex but only visits each vertex a second time and each edge once, so like BFS the algorithm still has big O notation run time as BFS at O(|V| + |E|).

## B. Edge Cutting

Cutting the edges is the important step to start separating the network into communities. The algorithm pushes the graph with the user specifying the desired number of communities the graph should have. It starts by ranking the edges based on its edge score sorted from highest to lowest. It then runs the community count algorithm checking to see if the number of communities is less than the desired number of communities. If community count returns a number lower than what the user provided it will remove the edge with the highest score. Edges will keep being removed until the desired number of communities or more are realized within the graph. This could happen until every edge is removed so, with community count being run at every iteration at a run time of O(|V|+|E|), the run time for edge cutting is O(|V||E| + $|E|^2$). Note that the search sort has a run time of O(|E|log|E|) which is lower than the rest of the algorithm and not necessary to be included in the final big O notation of run time.

## C. Additional reporting

Some additional algorithms were developed to help with the report. The community_size algorithm counts the number of vertices within each community returning a list where each spot is the equivalent community's name (which is a number) showing how many vertices make up each community. The community_vertex_list creates a list where each item is a sublist of all the vertices that are within each community and the initial list is one spot to the left community name (so community one is in the first spot or slot 0 of the list). Since the community's name is saved as an attribute within the vertex object via the community count and BFS algorithms, both algorithms have a run time of O(|V|).

## V. Look-A-Like Audiences

There are many use cases for utilizing the models explained to run community detection but for this paper we will focus on its use cases in marketing and social media networks. An understanding of what each stakeholder can gain from these algorithms requires some perspective from both the consumer and producer side of these models along with how social media networks can utilize it to identify desirable users and that these users become the product as they have monetary value to become identified for advertisers. This will assume that the ways that users are connected, such as being friends on Facebook or connected on LinkedIn, are the edges while each user is a vertex. An additional assumption is that these connections have significance especially to advertisers looking to target users that are part of the same community.

The social media networks that have these graphs available can utilize this for monetary means. Once they have the communities separated and identified, they can look at what attributes sell each of these communities as an ‘audience segment’, which is a list of users that have specific attributes marketers are usually looking for. There are also use cases for the site itself making the site more dynamic for its users. Specific communities might be happier in general with different features that are only available to them or turning off features that are not relevant to those specific communities. This allows for the users within those communities to have a better experience while using the website or social media network and will more likely spend more time on the site.

For marketers, they can leverage knowledge of these communities by utilizing them to target desired users with the goal of either selling their products or other important messages. One advantage to these models being unsupervised, or without prior knowledge of the users impacting the model, is that even if they have users that do actions highly desired by the marketer’s client, but do not know anything about the user other than their identity on the social media site, they can still identify users as part of a specific community within that network. This is known as ‘look alike modeling’(2) which is when a marketer provides a list of users and the social media network find users like that list. An algorithm was developed just for this.

The first step is utilizing the created userlist_to_objects function which transforms a user list, could be a name or id that can be mapped to the specific account of the user within the social network, into the equivalent vertex object. It then utilizes both the user_list and information about the communities within the network via the community count algorithm. It then scores the communities based on a customizable scoring algorithm based on three metrics:

- User Proportion: % of users in the user list that are a part of the community
- Network Proportion: % of users in the network’s community that are in the user list
- Combined Proportion: A combination of the user proportion times the network proportion

Once the communities are scored, it identifies the top scoring community and lists the users within the community vertex list algorithm. If they are not already in the advertiser’s user list, they will be added to a new audience list that gets returned by the algorithm. The algorithm has a set number of users to pull based on the marketer’s preference. If all the users within the top scoring community are included but not enough users that the marketer needs, it will go to the next top scoring community. This will continue until the desired number of users are pulled or until it goes through all the users within every community in the network. Given that it is only running through the vertices within the graph, the run time is $O(|V|)$.

## VI. Conclusion

With the community detection algorithm explained in further detail, social media networks can utilize the model to identify connections among their users and utilize these communities for specialized products and the creation of audience segments to use for monetary means. Marketers can utilize their current high valued consumers to target additional potential future customers through ‘look alike’ modeling by identifying users within the same community as their current user list.

Additionally, the model takes on a set number of communities removing edges until the number of communities appears in the network. There can be many other methods that could be done instead of the number of communities. One option is removing edges until the largest or smallest community is below or above a certain threshold or proportion of the total network.

There are a few caveats of the data that could be resolved with some edits to the algorithms. This model assumes that the weights are unweighted, but this is not always the case. For example, siblings might hold a much more significant connection than someone they went to elementary school with. The model could be optimized to give more weights, and lower scores, for edges with these stronger weights. The model also assumes the user knows how many communities the graph should have. This is something that almost never is known and there are many methods that could replace this including, but not limited to, edge cutting until one community is above or below a specific proportion of the total network or until specific attributes make up a large proportion of one community.

There are many other methods for finding community detection outside the one listed as discussed in the community detection documentation by Thamindu Jayawickrama(3). This algorithm is divisive meaning that it starts with the entire network and breaks down the network into the new communities. There are also agglomerative methods where each vertex is its own community and it groups vertices together until the communities are set up. Within these two methods are an array of different community detection algorithms each with its own methods and use cases for optimal communities. Two are the Leiden community detection which optimizes for discovering more weakly connected communities and the surprise community detection which works well for discovering many small communities. It is important to try out different methods to see which is optimal for whatever the desired use cases are for community detection.

## VII. References

(1)) pjoshi15, "Getting started with community detection in graphs and Networks," *Analytics Vidhya*, 13-Apr-2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/04/community-detection-graphs-networks/.

(2) "About Lookalike Audiences" *Facebook*. [Online]. Available: https://www.facebook.com/business/help/164749007013531?id=401668390442328+%283%29.

(3)T. D. Jayawickrama, "Community Detection Algorithms," *Towards Data Science*, 01-Feb-2021. [Online]. Available: https://towardsdatascience.com/community-detection-algorithms-9bd8951e7dae. [Accessed: 03-May-2022].