Rithika Devarakonda

1 May 2023

CSC 372: Seminar in Applied Algorithms

Professor Razieh Fathi

An Exploration in Creating an Optimal Random Community Detection Algorithm

Abstract

Communities are subgroups within a network structure that reflect nodes that share more ties amongst their group members than the rest of the graph. These communities are also reflective of users on social media in which those who have more shared interests or viewpoints are more likely to interact with each other and form tight knit communities leading to issues arising such as echo chambers and an increased likelihood of misinformation especially as these views become more harmful. My implementation, which is derived from the optimal community detection algorithm, introduces a random element in forming subgroups amongst a network in order to directly suppress algorithmic biases and the rise of echo chambers as forming randomized user hubs is more likely to offer a more diverse perspective and exchange of thoughts on social media. As such, the optimal random community detection algorithm can be potentially powerful when dealing with the rise of misinformation and centralized niche communities across social media as well as directly contradicting the personalized recommendations for alternate posts or influencers for any given user.

Introduction

My first experience with community detection algorithms was during my time taking the course: Social Networks, Theories and Methods at the Graduate Institute in Geneva last semester. Through that course, I was introduced to the world of Social Network Analysis, which is built upon the mathematical foundations of Graph Theory by adding an aspect of social dynamics and human interactions to every node and tie. Communities in the realm of social networks represent subgroups found within the larger graph based on attributes such as shared ties and a small degree of separation. In particular, I was working with the optimal community detection algorithm within California's House and Senate donor networks in which the purpose

of the group assignments was to create the most condensed and closely knit communities between various nodes to the point that there is a higher proportion of edges or ties between the set of nodes within their group compared to the total number of outgoing connections across communities. This proportion measuring the number of edges within communities compared to outgoing ties amongst the entire graph rather than within groups is called network modularity. The optimal community detection algorithm continues to assign nodes to various communities, creating different group divisions until the calculated network modularity score is maximized meaning we have found the most tightly connected subgroups in the entire network. Previously, I have used the built-in community detection algorithms R offers as part of the igraph library; however, during this project, I have chosen to work in Python instead.

My current research and implementation are based on the idea of returning the community assignment that is associated with the highest modularity score possible though I have chosen to add a randomized component instead. Specifically, my implementation creates subgroups on the assumption that the list of nodes is split based on a randomized index position meaning that a possible scenario could be that every node is assigned to the same community as the random integers or index position represent how many elements are included in every sublist. Likewise, every node could also be assigned to its own individual community. Regardless of group assignment, the worst case scenario is when the network modularity score of a given community assignment based on randomized subgroup slicing falls below 0 as that means that there is a higher amount of ties between nodes of different communities than the ones located within the same group. However, I also created another implementation in which the process of creating communities is iterated over a thousand times, and only the community assignment associated with the maximum modularity is output in which this result would dictate the most

optimal layout of the randomized optimal community detection algorithm I created. Now, I plan to delve deeper into the steps I took in developing this algorithm, starting from the first function called create_clusters() which takes in a graph's edge list of any size. My second function, as stated above, returns the community assignment of any graph with the highest modularity score, and thus the most optimal one named optimal_random() that also takes in a graph's edge list of any size.

Function 1#: create_clusters(edgelist)

Pseudocode:

```
def create_clusters(edgelist):
        e_list = every unique node in the given edgelist
        #create rand_list to contain the number of elements per every subgroup
        rand_list = []
        #now we generate a random number ranging from 1 to the number of nodes we have to determine how many nodes are included in every community and append them to rand_list
        output = the resulting communities that were split according to the random index position
        communities = output after removing every occurrence of an empty sublist
        modularity = built in networkx function that calculates the network modularity of our community structure generated from the communities variable
        group_assignment = []
```

```
        #the group_assignment list refers to the index of every community meaning
that we number them according to a given group's position amongst all of them
starting from community 0 to the last one where we have for example node A
associated with the first community
        return(communities, e_list, group_assignment, modularity)
```

The first part of my optimal random community detection algorithm takes in any given graph's edge list in order to return the community assignment of every respective node. The first step is creating a list of every unique node in the edge list to ensure there are no repeats when we continue to find random indices to split our edge list into subgroups. For example, many edge lists contain repeated nodes to express various ties, especially in larger graphs where nodes are more likely to have more than one tie. Thus, by finding every unique node it is possible to assign everyone to a respective subgroup. Next, we generate the same number of random integers as the size of the graph to ensure the possibility of a variety of subgroup distributions, in which case for a graph of size 4 the *rand_lis*t that contains the indices for the next step to split the node list on can be [2, 2, 2, 2] or [1, 4, 2, 3] where every integer represents the number of elements in every subgroup. After, we then formally create the new subgroups by separating the list of nodes, which are ordered according to their position in the given edge list, based on the number of elements the rand_list computed per every sublist to officially create our clusters. During this process, it's important to remove any empty sublist because of the random component implement, it is possible that the first entry is equal to the total number of nodes in which the first sublist contains every element in the graph. Thus, the remaining sublists, considering the *rand_list* is the same length as the number of unique nodes in the graph, have no more nodes left

to be assigned to any community. After removing any empty communities with no nodes assigned to them, we then calculate the modularity score of the community layout in which we create an entry displaying the subgroup assignments, a list of every unique node in the graph, the number associated with every subgroup such as group one or two based on the subgroups position in the overall communities list, and finally the modularity score.

Function 2#: optimal_random(edgelist)

Pseudocode:

```
def optimal_random(edgelist):
        #create empty list cases
        cases[]
        #now we generate a thousand iterations of community layouts from the
create_clusters() function and record every result in cases
        cases.append(first iteration of create_clusters(edgelist))
        for i in range(1000):
                x = iteration i of create_clusters(edgelist)
                #check to see if community structure is unique from previous iterations
                if x not in cases:
                        append x or  iteration i of create_clusters(edgelist) to cases[]
        max = the first modularity score associated with a community assignment as
recorded in the information cases[] stores
```

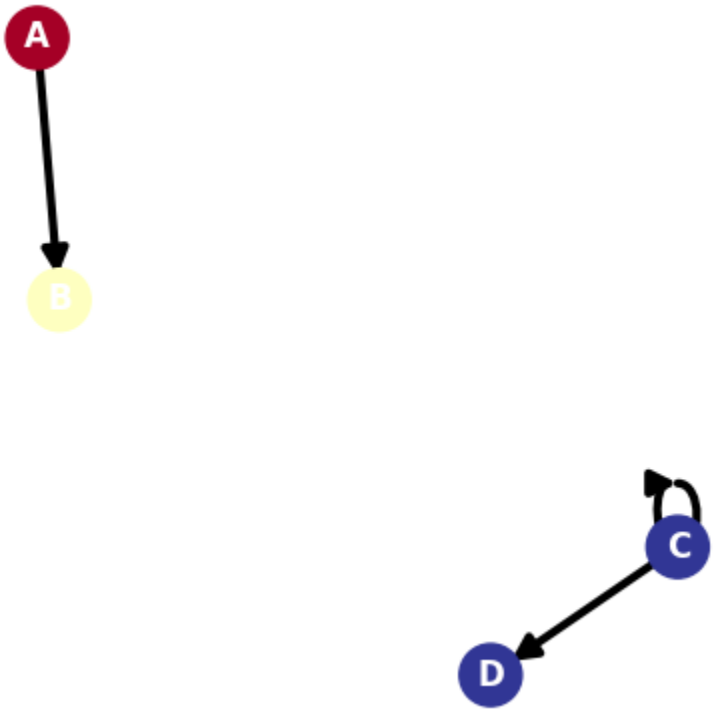
```
        #now we iterate through cases and determine the maximum network
modularity score and record the index (stored in the variable max_index) associated
with that value


        #our final result is the community assignment associated with the highest
modularity score otherwise known as the most optimal or tightly knit subgroups
amongst the overall network structure
        return(cases[max_index])
```

The second and final function takes in the same parameter as in the previous function create_clusters(), which is any graph's edge list. First, we call the create_clusters() implementation 1000 times in which we add every unique iteration to a list called *cases* to keep track of every result. Next, we loop through the *cases* list and specifically isolate the highest network modularity score recorded in any of the thousand iterations. Finally, once we have our maximum network modularity score, we return the associated results output from our create_clusters() as the index of whatever the maximum modularity score for the inputted graph edge list. The resulting communities and subgroup assignments output from this function represents the most optimal structural layout of communities found among the inputted graph's structure, which was our initial goal as part of the optimal community detection algorithm design.

Plots of Community Assignment Results

Though we can check the results of the algorithm by looking at the numerical distribution of subgroups or communities by seeing that node A is assigned to group 1, we can also represent these findings visually. For example, the number associated with every community can be made to fit a color as well as found in the plots below.




```
([['A'], ['B'], ['C', 'D']],
 NodeView(('A', 'B', 'C', 'D')),
 [0, 1, 2, 2],
 0.2222222222222222)
```

Figure 1. The output of create_clusters() on a small graph with nodes: A, B, C, D in which A and B were in their own communities while there was a group contained both C and D. Below is the data associated with how the communities were determine with the first

element representing the subgroups in the form of sublists, next represents every unique node in the edgelist, third is the associated numerical community for each node based on their positioning, and finally the last value is the network modularity score for the community structure.

Another thing to point out is that graphs can be represented by numbers or characters as well as that different edgelist compositions, despite containing the same nodal values: “R”, “I”, “T”, “H”, “K”, and “A,” can lead to different community detection results on the basis of optimal subgrouping being based on ties between the various nodes.

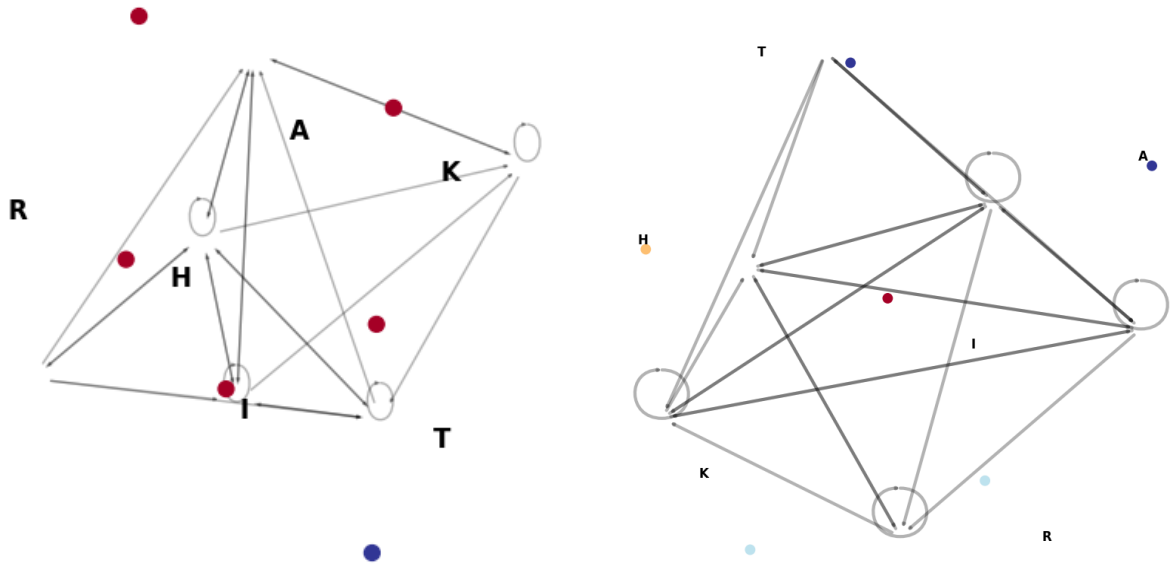


Figure 2. A comparison of how edgelists containing the same nodal values can produce different results in how communities form on the basis of different ties being present along with the underlying random component as well as how the leftmost graph was generated by one iteration of the create_clusters() function while the rightmost one was created using the

results of the optimal_random() implementation that takes into account a thousand runs and uses the best performing community distribution.

Finally, the optimal random community detection algorithm can also work on large graphs of over a hundred nodes along with the smaller ones as shown previously.

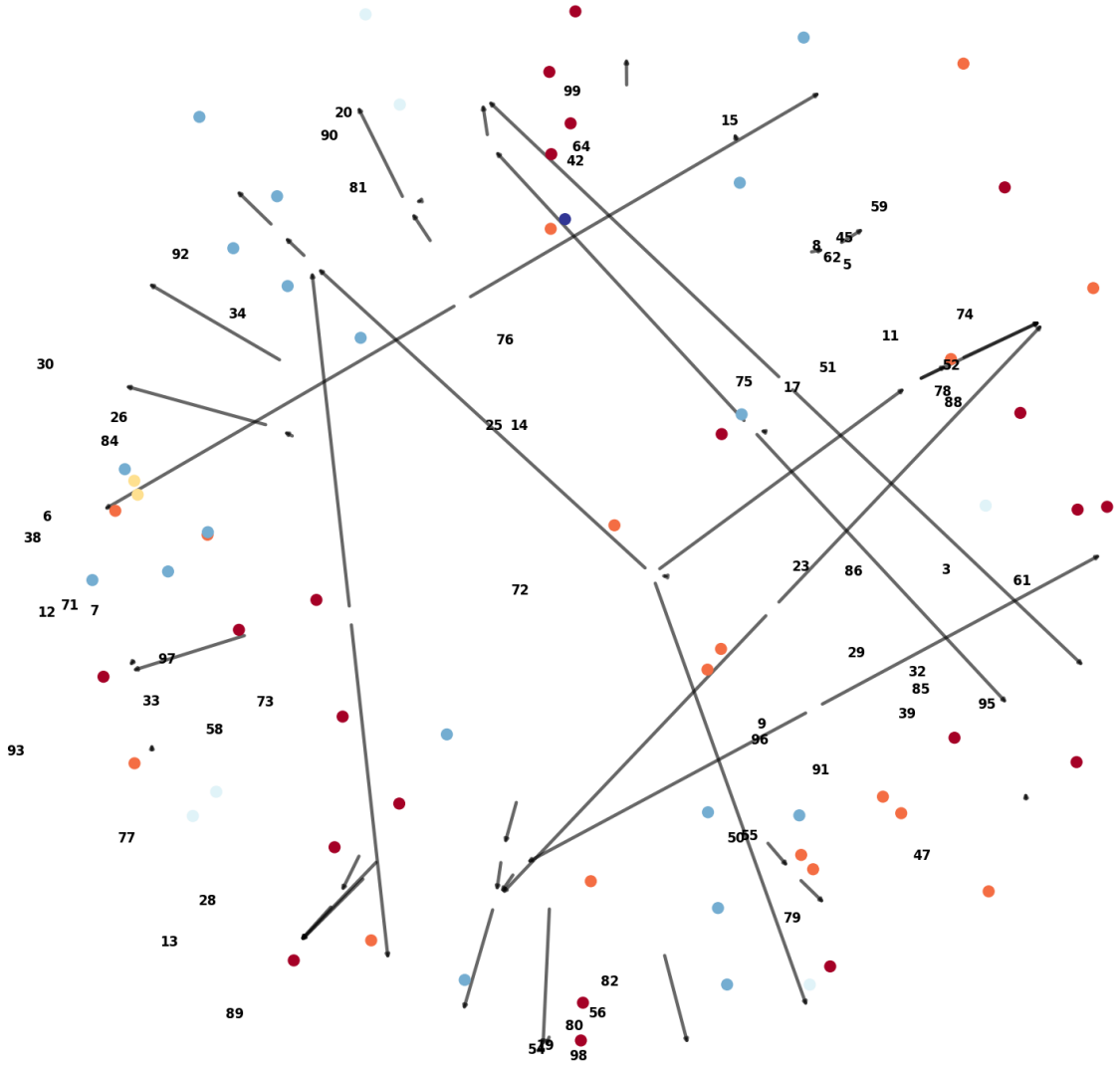


Figure 3. The result of the optimal_random() implementation on a graph that contained over a hundred nodes with the most cohesive subgroups detected amongst the large network structure.

Uses for the Optimal Random Community Detection Algorithm

Users on social media form their own networks and thus inherently belong to communities in the sense that the majority of people interact with a limited portion of the entire user base for any app or website. As Bedi and Sharma state in their paper "Community detection in social networks", "The tendency of people with similar tastes, choices, and preferences to get associated in a social network leads to the formation of virtual clusters or communities. Detection of these communities can be beneficial for numerous applications such as … finding a set of likeminded users for marketing and recommendations" (Bedi and Sharma 1). Thus, once users continuously interact with specific content tailored to their interests they fall into more niche and tight-knit communities, which also leads to the rise of echo chambers and an increased likelihood of spreading misinformation due to the lack of diverse perspectives in these communities especially if the majority of the users in these hubs adopt or repeat hateful biases and discriminatory ideas. However, the optimal random community detection algorithm is a tool that can suppress the pipeline of misinformation, hate groups, and echo chambers by introducing a randomly generated set of communities that contain users. This randomization of subgroups in social media can create a more diverse set of users, who can interact with each other as well as offer more perspectives and different recommendations leading to a lower likelihood of misinformation spreading along with suppressing algorithmic biases that directly hurt marginalized communities. For example, content created by marginalized creators and

discussions or talks held by them can be recommended and shared with a wider audience of randomly selected or generated communities as a way to directly contrast with the algorithmic bias present in current social media that caters to the perspective of privileged communities.

Conclusion and Further Work

Overall, I started my research on writing a version of a community detection algorithm from scratch due to my interest in understanding what goes on during the process of using the built-in functions in R-based libraries such as igraph. However, I realized that adding a randomized component in determining where subgroups of an edge list were sectioned off would be an interesting way of comparing the results of my implementation with the built-in function's version to see if there was any difference as such. Before adding my function optimal_random(), I was not quite satisfied with the results of the create_clusters() implementation alone because oftentimes I could tell that the randomization ensured that running the community detection algorithm once would not return the most optimal or best version of the group assignments. Thus, after adding in the iterative component as well as keeping track of every unique result and only returning the community layout associated with the highest modularity score, I feel that my implementation would return the same results as the built-in optimal community detection algorithm.

On the other hand, a weakness that the randomization exploits is the issue of time and storage as I believe that my implementation's time complexity is much higher than that of the built-in community detection algorithms. Because of the random component, a user would have to iterate through the process several times more than if they used the built-in function which would take up more time and space on someone's device especially as the graphs they input

contain more nodes. Going forward, I plan on implementing the steps of the Breadth First Search algorithm where we search for nodes of every depth and add one at every level into the community our starting node is in and then calculate the network modularity score at every given addition of our subgroup structure. Thus, I think that using the idea of searching for the most optimal path between nodes in a graph might be an interesting way in order to incorporate how that aspect relates to how nodes are added into subgroups or removed iteratively in order to find the most efficient community-based graph structure. This method might prove to be more time efficient, so moving forward I plan to find an alternative means of splitting a graph into various subgroups or communities to find more ways of replicating the process of the optimal community detection algorithm whilst also considering the best case use of randomization.

Resources Consulted

Bedi, Punam, and Chhavi Sharma. "Community Detection in Social Networks." *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 6, no. 3, 19 Feb. 2016, pp. 115–135., https://doi.org/10.1002/widm.1178.