Strongly Connected Components And Social Media Networks
Karthik Shanmugam

Introduction

Social Media Networks are used to describe the relationship between various entities on a social media platform. These networks typically leverage graph theory and properties of graphs to gain some insight into the kinds of relationships modeled. In this project, I aim to explore a particular property of such graphs called strongly connected components, and apply this concept to a common problem in social media networks : Advertisement targeting (Community detection).

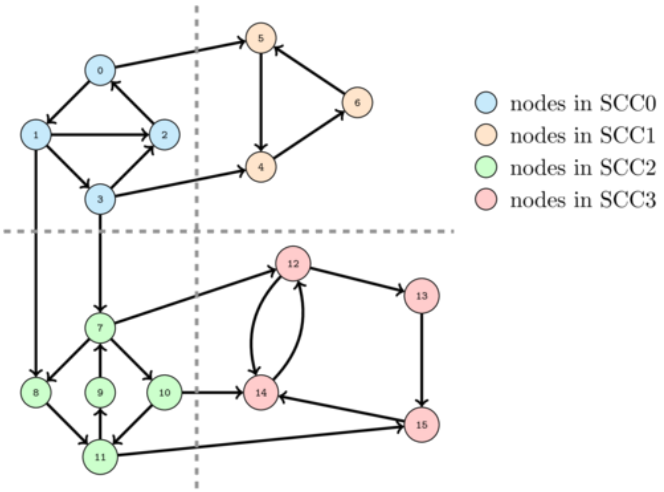
Strongly Connected Components

Typically, in the context of social media, each node in the graph represents either a person or a "page" within the network. An example for a page could be ESPN, which is a sports news channel with its own page on the platform.

The graphs that model these networks are directed graphs, since the direction of the relationship between any two nodes (people) is important. In this context, if person A follows person B, and person B in turn follows person C, and C follows A, the resultant graph contains a cycle. Cycles between nodes indicate that these nodes have a lot of connections between them. In a way, they are "strongly connected".

More formally, a strongly connected component is defined as

"A strongly connected component of a directed graph G is a maximal set of vertices C ⊆ V such that for every pair of vertices u and v, there is a directed path from u to v and a directed path from v to u."



Example Graph 1.0

Kosaraju's Algorithm

1. Do a DFS on the original graph: Do a DFS on the original graph, keeping track of the finish times of each node. This can be done using a stack, when a DFS finishes put the source vertex on the stack. This way the node with the highest finishing time will be on top of the stack.
2. Reverse the original graph: Reverse the graph using an adjacency list.
3. Do DFS again: Do DFS on the reversed graph, with the source vertex as the vertex on top of the stack. When DFS finishes, all nodes visited will form one Strongly Connected Component. If any more nodes remain unvisited, this means there are more Strongly Connected Components, so pop vertices from top of the stack until a valid unvisited node is found. This will have the highest finishing time of all currently unvisited nodes. This step is repeated until all nodes are visited.

PseudoCode :

```
stack STACK
void DFS(int source) {
    visited[s]=true
    for all neighbours X of source that are not visited:
        DFS(X)
    STACK.push(source)
}

CLEAR ADJACENCY_LIST
for all edges e:
    first = one end point of e
    second = other end point of e
    ADJACENCY_LIST[second].push(first)

while STACK is not empty:
    source = STACK.top()
    STACK.pop()
    if source is visited :
        continue
    else :
        DFS(source)
```

Complexity Analysis:

Kosaraju's algorithm takes similar time complexity as Depth First Search (DFS). It is equivalent to the sum of vertex and edges. I does a two-pass depth first search so it takes twice the time, along with some additional time to reverse the graph edges, but still the time complexity is linear.

Time Complexity : **O(V+E)**

In terms of space complexity, it has a space complexity of O(V), which is equal to the number of vertices on the graph since the stack is used for the same.

Application : Social Media Advertisement Targeting

A common requirement in any social media platform is to publish advertisements.

Advertisements cannot be published and broadcast to everyone on the network. There needs to be specific segments of the people on the network that will be shown the advertisement, based on the fact that these people are more likely to find it useful. This is formally called targeted advertisements. There are different ways of targeting based on segmentation, demographics etc. Here we will explore how SCCs can help in targeting advertisements.

Setup:

The network has two basic types of nodes: People and Pages, each object can be modeled as follows :

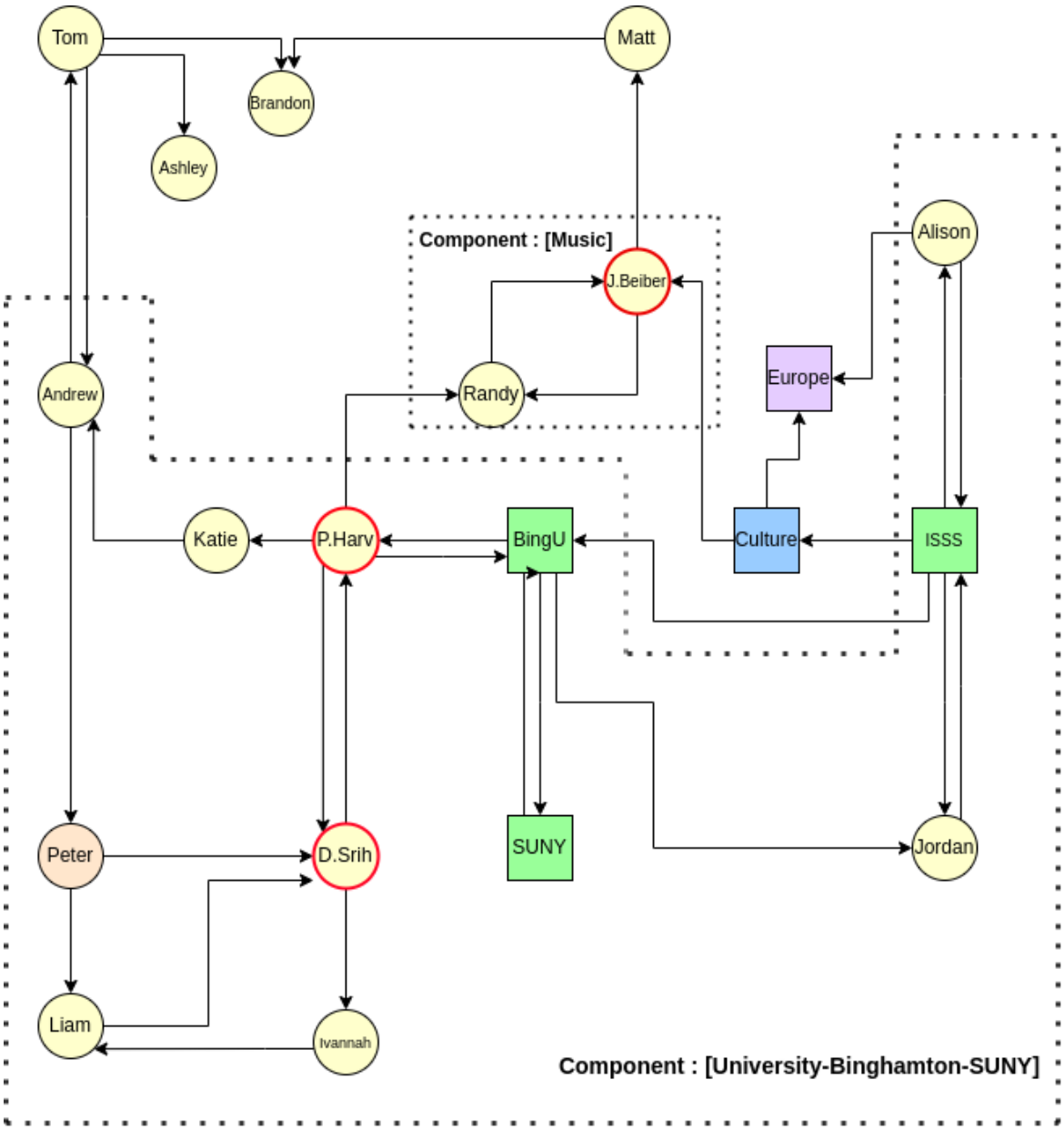
```
class Node:
   def __init__(self, value, type, attr):
       self.value = value
       self.type = type
       self.attr = attr
```

```
People :  {value="Alice", type = "People", attr="{Age: 22, Sex: F}"}

People (Influencer) :  {value="Michael Jackson", type = "Influencer",
attr="{Age: 22, Sex: F, Tags:{"Music", "Rock"}, Weight: 4000}"}

Pages :  {value="Binghamton University", type = "Page", attr="{Age: 22, Sex: F,
Tags:{"Music", "Rock"}, Weight: 200}"}
```

Certain People nodes (Influencers) and Pages have additional attributes such as Tags and Weights. Every page will have a tag describing what that Page is about. Each Page will also

have a Weight, based on the degree of the node - this weight basically tells us how popular (or influential) this node is. This will be important when we tackle a different problem called Component Labeling.

Consider an example network as shown below. It has many People nodes (circular) and Page nodes (squares). Some People nodes have red circular borders. These are termed “Influencer” nodes and we will see their significance in the next section. This network shows certain pages that are related to Binghamton University : ISSS, BinghamtonUniversity, and SUNY. There are also other Pages such as Culture, Europe etc. All other nodes are people nodes, and the connections between these nodes indicate that they follow each other.



Example Graph 2.0

When we run an SCC detection algorithm on this such as Kosaraju on this graph, we get the following components (Implemented program output) :

```
Component [ -University-Binghamton-SUNY- ] :
-----------------------------------------------------

SUNY ( Page )
Katie ( Person )
Tom ( Person )
Andrew ( Person )
Peter ( Person )
Ivannah ( Person )
Liam ( Person )
D Srihari ( Person (Influencer) )
Harvey Stenger ( Person (Influencer) )
BingU ( Page )
Jordan ( Person )
BingISSS ( Page )
Alison ( Person )

Component [ -Music- ] :
-----------------------------------------------------

Justin Beiber ( Influencer )
Randy ( Person )
```

Here, we can see that there is a large component related to Binghamton university, and a smaller one related to the two node components (labeled music). By default, every single node is a component itself, but we will be ignoring those as they are not relevant to this problem. The words within the square brackets next to each component is the component's label.

Component Labeling

Component labeling is important because that is the only way to give meaning to the group of nodes on the network. In order to use these components in a meaningful way, we need to capture the common attributes between all the nodes within the component.

In order to make this accurate, I designate some nodes as influencer nodes. These nodes have a high number of incoming and outgoing edges. In terms of graphs, these nodes have a high "degree". These influencer nodes typically have a greater influence on the final label of the component, since it makes sense intuitively. If there is a component with influencer nodes such as Shakespeare and J.K.Rowling, it makes sense that this component most likely has pages related to books, and people following these pages are book readers.

In the example above, certain nodes have a red border. Example, P.Harvey (President Harvey) and D.Srih (Dean Srihari) are such highly popular influencer nodes.

Influencer nodes and pages have Tags and Weights. Each tag is a string describing the node. For example, a page "SUNY" may have tags "New York", "University", etc.

The Weights on the page and influencer nodes depend on the degree of the node. A relatively less popular page such as SUNY will have a lower weight than for example an extremely popular celebrity such as Michael Jackson. Taking all this into account, we can have a simple algorithm to take the largest weighted nodes from the component subgraph and derive a short label that summarizes the characteristics of the group (and component).

Algorithm:

1. Compute all nodes on the graph with high degree (large number of incoming and outgoing connections) and store these in an array : iNodes.
2. Select all the Pages in the graph and store these in the Pages array.
3. For each iNode, Page assign a weight based on its degree, store these weights in Win and Wp arrays.
4. Sort decreasing (Win), Sort decreasing (Wp)
5. Based on highest weight, use that particular weight or page's [Tags] array to add the tag to the final label.

Implementation (weights hardcoded):

```
# Component Labeling Algorithm
for j in range(len(scc_output)):
   compName = ""
   maxWeight = -1
   for v in scc_output[j]:
       if(v.attr.get("Weight") != None):
           sorted(v.attr.get("Weight"), -1)
           if(v.attr.get("Weight")>maxWeight):
               maxWeight = v.attr.get("Weight")
               cNm = ""
               for i,e in enumerate(v.attr.get("Tags")):
                   cNm = cNm+"-"+e
               compName = cNm

   if(compName == ""):
       compName = "-Default"
   compNames.append(compName+"-")
```

Time Complexity: O(N * NlogN) where N is the number of Vertices.

In order to sort the weight arrays, we need NlogN time. We need to go through each node in each component and assign a label to it. In doing so, we go through every node in the graph. If there are N nodes, we go over it N times.

Therefore : N (check tag based weights) * NlogN (sort) => **O(N*NlogN)** is the time complexity.

Once we have Components and Labels, it is easy to see how we can use this for advertisement targeting. If for example, we have a book publisher like Barnes & Noble who wants to publish ads for a new textbook, we can simply match with component labels such as ‘University’ and broadcast the advertisement to every node within these components.

We can go further and match with ‘Binghamton’ and ‘University’ to refine the nodes which are targeted.

The process of identifying components and labeling them can be done at many levels. Components themselves can be considered as nodes and then super-components can be formed and labeled and so on.

Outside the context of advertisement targeting, this approach can be used for general community detection in social media networks.

Note on Program Implementation: The practical code implementation for this project is based on this application. It uses Kosaraju’s algorithm as well as the component labeling algorithm to generate components with labels for the example graph (Example Graph 2.0 used above). To run the program, make sure python3 is installed, then go to the directory containing the SCC_SMN.py file and execute the command:

```
python3 SCC_SMN.py
```

***

References:
[Example Graph 1.0]
https://www.researchgate.net/figure/Strongly-connected-components-of-the-example-graph_fig1_326145722