Graph Partitioning Using Betweenness Measures in Distributed Environment

Shiyang Cheng(SC57227), Cong Wang(CW37657),
Marina Thomas(MT34799), Kyle Sung(CS9893)

Abstract
This project implements a version of the Girvan-Newman algorithm used for finding communities in a network that can be run in a distributed environment, where each process is represented as a node in the network. The Girvan-Newman algorithm uses betweenness centrality to measure the importance of an edge in the network and gradually removes the important edges until the network is divided into two or more communities. The distributed environment consist of processes that are only aware of immediate neighbors. The algorithm also assumes reliable FIFO network. Each node is allowed to broadcast to all the nodes in the network. The project is evaluated by counting the messages required to divide the network graph into two partitions.

1. Introduction

Many systems can be considered as a graph which includes vertices and edges. In computer science world, there are two typical kinds of networks: social network, Internet of Things device networks. Typically people are interested how many groups in the graph to understand the structure of the graph and to analysis the relationship between each group of vertices. In social network, partition could bring some value to sell advertisement. In the IoT network, it could help to analyze the distribution of the devices and their connectivity.

Researcher already proposed some brilliant approaches to address graph partition problem. Girvan-Newman proposed a method exploit the flow and betweenness of the graph to divide the graph to several subgroups. Vertices in each subgroup get more connectivities within the subgroup, but get less connectivities between vertices in other subgroups. To identify these subgroups from a graph is the definition of graph partitioning problem. This paper only focuses on partition graph based on betweenness approaches.

Recently, the graph size becomes larger and larger. For example, facebook is a huge social network with more than 2 billion users. Original version of Girvan-Newman method is designed to be processed on one machine which hardly hold the whole data. Data is naturally generated in Internet of Things devices, which is hard to directly apply Girvan-Newman method.

This paper proposes a distributed algorithms to handle the data spread into different nodes. This algorithm could run in parallel between each vertices. To keep all vertices in the graph, this paper only consider remove edge without remove any vertex.

2. Description of the project

In our environment we assumes that the nodes in the network graph exist on the same physical network, and the nodes can send messages to their immediate neighbors on the network graph, or they can broadcast to every single node in the graph. The nodes do not have shared memory and messages can arrive with arbitrary delay. However we do assume that the the communication channel is reliable and preserves FIFO ordering, and the process on each node is reliable.

The limitation of a distributed environment complicates the implementation of Girvan-Newman algorithm because the nodes cannot be visited in a breadth first search sequence. The first step of the algorithm requires each node to calculate the minimum distance from itself to every other node. Because the message delivery timing can vary, a node might need to frequently update its minimum distance to all its neighbors, and termination can be hard to determine without careful design. However, there are existing algorithm for breadth-first search that on a distributed environment that can reliably calculate the shortest path distance between nodes and determine the termination of the execution [1]. This project expands on the idea and applies the algorithm to calculate betweenness.

3. Our Algorithm

Before we move into the details of the algorithm, below are some of the details of the system considered.

<u>Processes involved</u>
Initiator: Equivalent to ‘Environment’ used in Dijkstra’s algorithm for termination detection.
Node: Each server that is part of the distributed network considered.
<u>System Knowledge</u>
Initiator: Knows details to reach each server
Node: Knows how many servers are in the system. It’s own neighbors and details to reach its neighbors. Its own ID.

<u>Messages</u>
initiate: Sent by Initiator to initiate the betweenness calculation
path: Initiated by each root node and propagated by every node to its neighbor to find the shortest path to the root node.

marker: Sent from every node to its neighbor to indicate the completion of the task it was asked to do.
DoneInitiate: Sent by root node to its neighbors to indicate that it is now done initiating shortest path calculation.
parentValue: To initiate parent value calculation
parentValueMarker: to indicate parentValue calculation complete
Betweenness: to calculate betweenness

The calculation is done in 3 phases:

1. Calculate the shortest path from each node to all other nodes using diffusing computation.
2. Determine the value of each node in each shortest path.
3. Considering each node as the root node, determine the amount of flow from the root node to all other nodes using each edge.

Pseudo code for each of the above step is explained below:

A special process called ‘Initiator’ initiates the process, by calling each server with the message “initiate”.

1. Shortest path from each node to all other nodes using diffusing computation. The pseudo code is below

```
Var

ID: Node id
deficiencyMap: A map that contains deficiency, integer initially 0, for each
beginningNode flow;
parents: integer array list, process ids initially null;
shortestPathsDistance: array of integers to track the shortest path from current node to
root. initially the distance from each node is the MAXINT

Upon receiving a "initiate" message from Pj for beginningNode k:
callNeighbors(); //pass current distance as 0

Upon receiving a "path" message from Pj for beginningNode k:
    if (shortestPathsDistance[k] > DistanceFromPj+1) then
        sendMarkerToParents()
        parents = new list;
        parents.add(Pj) ;
        shortestPathsDistance[k] = DistanceFromPj +1;
        callNeighbors(); //for shortest path calculation
    else if (shortestPathsDistance[k] == DistanceFromPj+1) then
        if parents contains Pj then
            send marker to Pj
        else
            parents.add(Pj)
            else
                    Send marker to Pj

    CallNeighbors(beginningNode, currentDistance):
            For each neighbor:
                    Increment deficiency for the beginning node
                    Call Neighbor

    Upon receiving a "marker" message from Pj for beginningNode k:
            Decrement the deficiency count for that beginningNode
            If deficiency ==0:
                    If beginningNode != ID:
                            Send marker to neighbors along with the farthest distance I know
from the beginning node
                    Else:
                            // Shortest path distance calculation done
                            Add my ID as done
                            If the distance received is < the distance travelled in previous round,
                            termination.
                            Else: Call neighbors with message "DoneInitiate" and ID of all
                            nodes that I know are done.

    Upon receiving a "DoneInitiate" message from Pj for beginningNode k:
            Store the IDs of all nodes that are done.
            If all nodes are done, initiate parent value calculation
```

2. Determine the value of each node in each shortest path.

```
Var

parentValue; integer

initiateParentValueCalculation
    Call neighbors with parent value as 1
    Increase deficiency count per neighbor

Upon receiving a "parentValue" message from Pj for beginningNode k:
      myParentValue = number of parents to the beginning Node
      Check if any neighbor who is not my parent can be called.
```

```
                If yes, call neighbor;
                        Increase deficiency count
                If not send parentValueMarker along with combined parent value of the
        node and its children to parent

        Upon receiving a "parentValueMarker" message from Pj for beginningNode k:

                    Decrease deficiency count
                    Parse and save parent value;
                     If deficiency ==0:
                        If beginningNode != ID:
                                Send combined parentValue of me and by children to
parents
                        Else:
                               // Shortest path distance calculation done
                               Add my ID as done
                               Call neighbors with message "DoneParentValue" and ID of
all nodes that I know are done.

            If parentValue of all nodes known:
                 Initiate_Flow_Value
```

3. Initiate flow value/ Calculate betweenness.

```
Find leaf nodes by looking into parentValueString
Add leaf node to a queue
While queue not empty:
        queue.pop
        From the leaf node calculate the betweenness string using the below
        Leaf node has an incoming flow of 0
        For each parent
              Add parent to the queue
              Get parent value of parent
              Betweenness = (1 + incoming flow)*parent's parent value/myParentValue
              incomingFlow of another node = Betweenness + current incoming flow
              Track the betweenness in an mXm matrix where m is the number of servers

Once betweenness is calculated, find the edge with max betweenness
If the edge is part of the current server, remove the neighbor

Do initiate flow again
```

3.1 The Code
Git Hub URL : https://github.com/UT-APT/SC2018_Proj

3.2 Validation Part

To validate our algorithm, we use an existing network analysis package - igraph to help us achieve this goal. Igraph is a collection of network analysis tools with the emphasis on efficiency and portability and ease of use. And igraph is open source, it can be programmed in R, Python and C/C++. So we use python-igraph package to test the graph in our textbook.

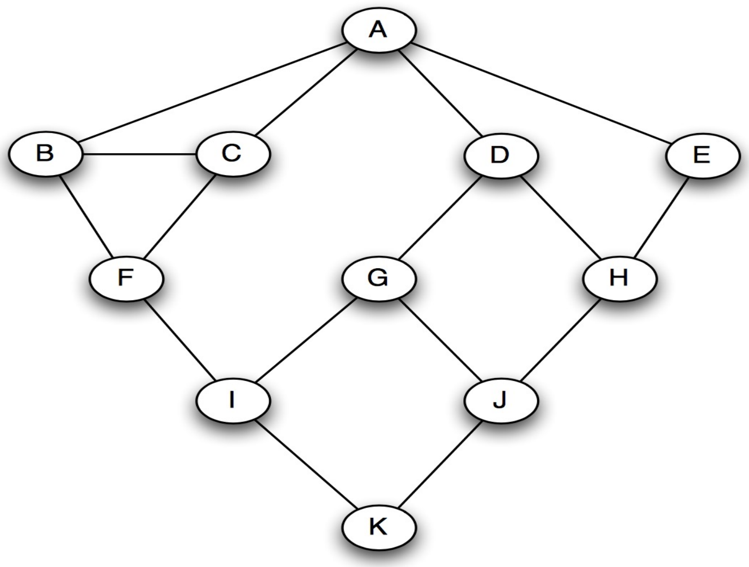

Figure 1 test example

We first need to write the graph as an edge list, for example, as in Figure 1, node A has neighbour B, C, D and E, so in our edge list, we have (A, B), (A, C), (A, D) and (A, E).
1. And as we get the input edge list, after getting the edge betweenness with function edge_betweenness(), we get the result : [7.39, 7.39, 10.75, 8.02, 1.0, 6.19, 6.19, 8.88, 6.50, 7.35, 11.90, 8.52, 6.11, 8.75, 7.38, 6.71] . Because the igraph reads a file and create isolated vertices, that save the original IDs in a vertex attribute named name and let the vertex IDs be consecutive. So we get the highest edge betweenness (5,8) - (F, I);
2. Then we manually delete edge (F, I) with delete_edges(edgeID), and calculate the edge betweenness again. The result is [12.0, 12.0, 20.75, 9.42, 1.0, 5.0, 5.0, 16.50, 8.08, 9.25, 9.75, 6.58, 11.50, 3.92, 8.25]. The highest edge betweenness is (0, 3) - (A, D);
3. we manually delete edge (A, D) again, calculate the edge betweenness, whose result is [12.0, 12.0, 28.0, 1.0, 5.0, 5.0, 8.17, 11.83, 30.0, 7.33, 7.83, 19.83, 4.33, 10.67]. The highest edge betweenness is (4, 7) - (E, H)

4. Now as indicated by igraph, the graph has been partitioned into two clusters, they are (A, B, C, F, E) and (D, G, H, I, J, K).
5. When we use our algorithm to read the input files, the result show as below:

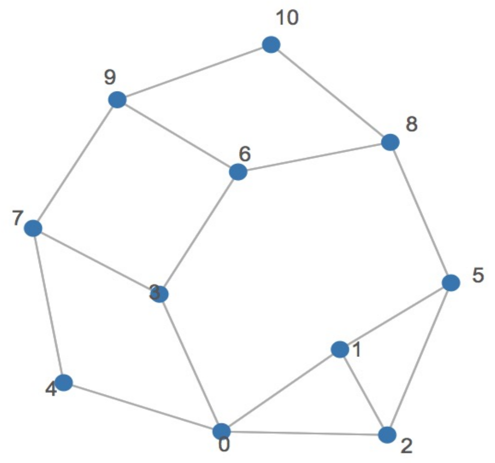

Figure 2 original test

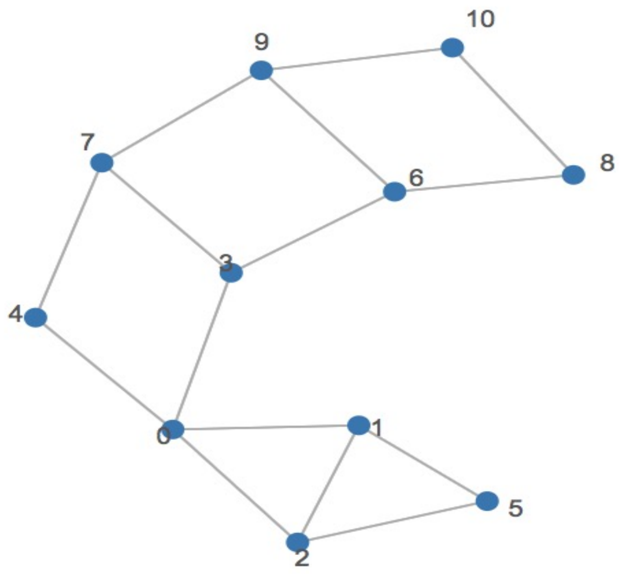

Figure 3 first round of computation, edge (5,8) that has the highest edge betweenness is removed

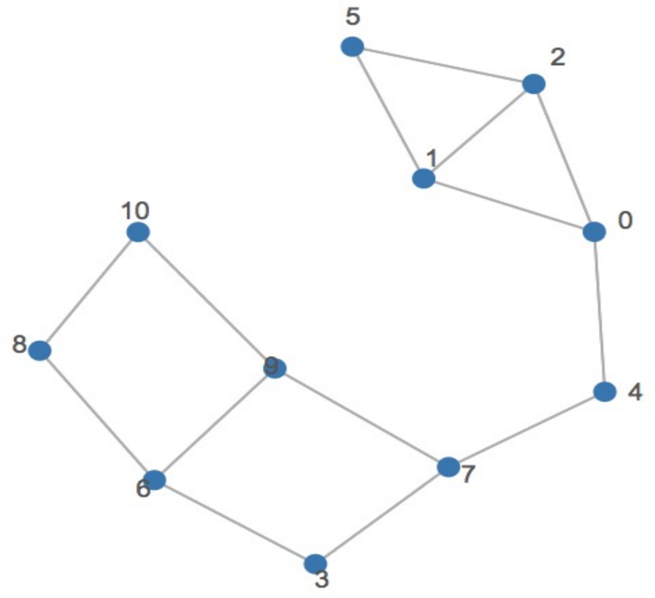

Figure 4 second round of computation, edge (0, 3) that has the highest edge betweenness is removed

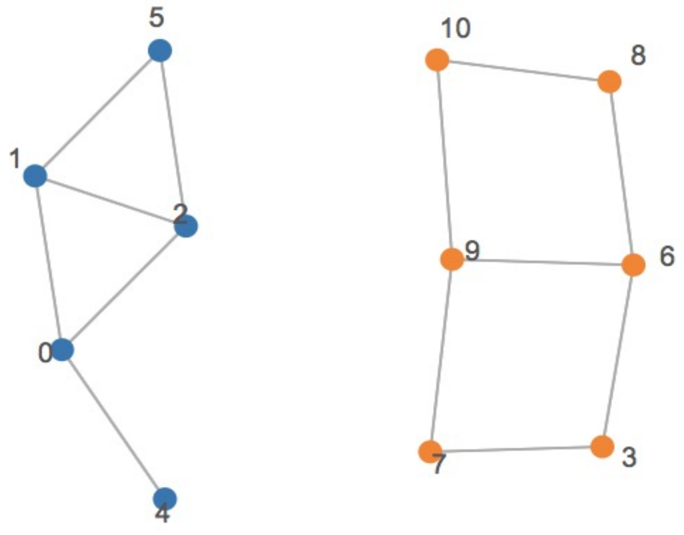

Figure 5 third round of computation, edge (4, 7) that has the highest edge betweenness is removed

Generally edge betweenness gives the most satisfying results in all community detection algorithms, but it is a pretty slow method. The computation for edge betweenness is pretty complex and it will have to be computed again after removing each edge. It produces a dendrogram with no reminder to choose the appropriate number of communities. But for igraph it does a function that output the optimal count for a dendrogram.

5. Message complexity

Graph	Nodes	Edges	Message Count
4-in-a-row	4	3	164
barbell	6	7	610
A-K example graph	11	16	2792

6. Conclusion

Our implementation of the distributed Girvan-Newman algorithm showed that it is possible to perform betweenness-based graph partitioning in a distributed environment. However, the implementation as described in this report scales poorly as the number of nodes increased. Further optimization of the messages might be possible - the current implementation has three phases of messages in each round of finding the edge with the highest betweenness, and each phase requires a round trip of messages and ack messages. For example, the betweenness message travels from leaves to the root, so it might be possible to combine it with the ack messages from the previous phase. But that is out of the scope of this project.

7. References

[1] Dijkstra, Edsger W.; Scholten, C. S. (1980), "Termination detection for diffusing computations" (PDF), Information Processing Letters, 11 (1): 1–4, doi:10.1016/0020-0190(80)90021-6, MR 0585394.