Detecting Communities In Graphs with Highly Overlapping Communities using Bloom Filters

Pratham Shah Shubham Jain

February 12, 2022

Abstract

Community detection is an interesting problem in the area of social networks. It gives us useful insights into the underlying structure of complex networks. Detection of communities in graphs can be a challenging task, and it poses an even bigger challenge if the graph has overlapping communities. In this report, we present an algorithm which detects local communities in graphs with overlapping communities. We use Bloom Filters to reduce memory footprint on the main memory.

1 Introduction

Community in a network is a subset of nodes that are more densely than the rest of the network. Community detection is an interesting problem in the area of social networks. It gives us useful insights into the underlying structure of complex networks. There exist algorithms which give good results in finding communities in graphs when the communities are not overlapping [LFK] [ACL08]. However, community detection is a challenge when the graph has overlapping communities.

In this report, we present an algorithm which detects local communities in graphs with overlapping communities. We use Bloom Filters to reduce memory footprint on the main memory.

2 Preliminaries

2.1 Bloom Filters

Bloom Filters [Blo70] were introduced by B. H. Bloom in 1970. It is a space-efficient data structure that answers set-membership queries with some false positive rate. They have found numerous applications due to their tremendous space-efficient nature.

2.2 Bloom Sample Trees

A Bloom Sample Tree [SBBR] is a novel data structure that can be used to perform multiple operations involving Bloom Filters. A Bloom Sample Tree organises the namespace like a binary search tree. Each node is a Bloom Filter that contains a subset of the namespace. Union of all the Bloom Filters at a level yields the entire namespace. We use Bloom Sample Trees here to sample multiple elements from Bloom Filters and to calculate the set difference between two Bloom Filters.

2.2.1 Sampling From Bloom Filter

We extend the method provided in [SBBR] and use threads to sample multiple elements from a Bloom Filter.
Given a Bloom Filter B to sample n elements from, we proceed in the following recursive fashion to get n samples elements:

- Create a Bloom Sample Tree with the namespace of the Bloom Filter.
- At a given (non-leaf) node, compute the intersection of the Bloom filters stored in the left and right children of this node with b. If for both child nodes, the intersection is empty, then

b does not contain any element belonging in the range associated with this node. Therefore the subtree rooted at this node is pruned from the search.
The estimated number of elements in the intersection of two Bloom filters $\mathcal{B}_1$ and $\mathcal{B}_2$ is given by the following expression [PSN10]:

$$\hat{S}^{-1}(t_1, t_2, t_\wedge) = \frac{\ln\left(m - \frac{t_\wedge \times m - t_1 \times t_2}{m - t_1 - t_2 + t_\wedge}\right) - \ln(m)}{k \times \ln\left(1 - \frac{1}{m}\right)}$$

where t_1 is the number of bits set in $\mathcal{B}_1$, t_2 is the number of bits set in $\mathcal{B}_2$, m is the size of both Bloom filters, k is number of hash functions used in both, and $t_\wedge$ is the number of bits set in the bitwise AND of $\mathcal{B}_1$ and $\mathcal{B}_2$.

- If intersection with only one child is non-empty, then a new thread is created to proceed the search along that child node. The other child node and the subtree rooted at it are pruned from the search.

- If intersection with both child nodes is non-empty, then two threads are created, one for proceeding the search along the left subtree and the other to carry the search along the right subtree. The number of elements to be sampled from the right and left child nodes is proportional to the number of elements in their respective intersections. Note that it is possible that the intersection was a false positive and this is discovered further down this subtree. In that case, the search then backtracks and proceeds along the other child node.

- At a leaf node, every element in the range of the node is checked for membership in b. The samples at this leaf node are sampled uniformly at random from the set of values that satisfy the membership test of b. If none of the elements within this range satisfies the membership query, it indicates that the search has reached this leaf node due to a (string of) false set overlap(s). In this case, the sample at this node is $NULL$.

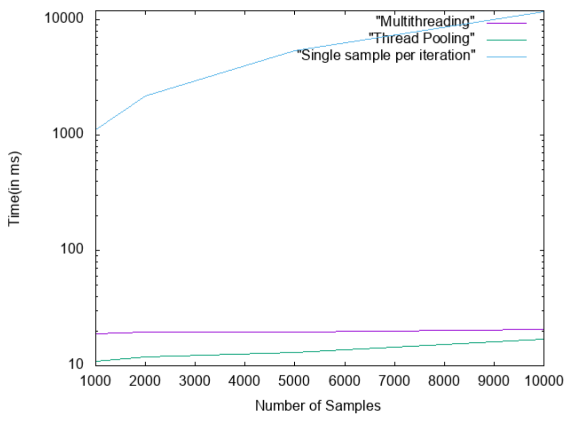
One another approach to sample elements from a Bloom Filter using a Bloom Search Tree is to create a thread pool. A fixed number of threads are created. To sample n elements, a job queue is created, and the task is added to the job queue. Whenever a thread is idle in the thread pool, it fetches a job from the job queue and samples an element independently.

Experimental Analysis We used a static synthetic namespace to compare the running time of the different approaches. We varied our namespace from 10^5 to 10^7. We also varied the number of samples to be generated from 1000 to 10000. In the combined approach, we have used 256 threads for the below experiments. In the experiments we performed; we generated query sets uniformly at random from the entire namespace. We compared the running time of both the approaches listed and compared it to the running time of generating a single sample, the required number of times.

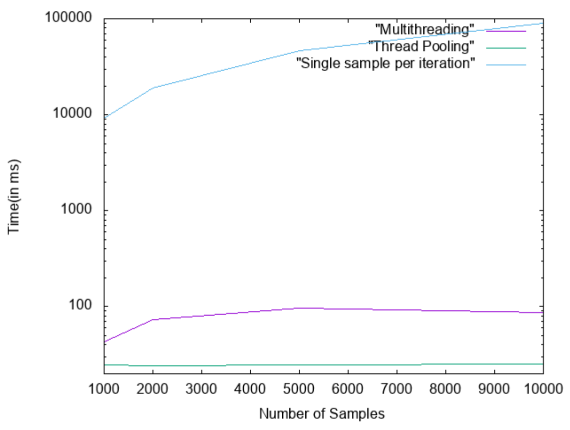
Results Figure 1 shows the average time taken to sample multiple elements from a set using a Bloom Sample Tree using different approaches. The experiment demonstrate that creating multiple threads in advance and then using each to handle multiple nodes is the best strategy of the proposed ones. $Multithreading$ refers to the approach where we create a new thread at each node and distribute the task among children of a node. $Threadpool$ refers to the approach where we create 256 threads and use a job queue to sample elements.

2.2.2 Calculation of Set Difference Between Two Bloom Filters

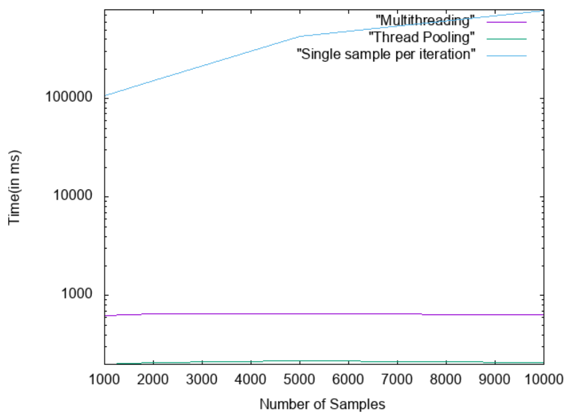
In this section, we give a method to estimate elements in the set difference between two sets represented as Bloom Filters.
Problem Statement. If we are given two sets $S1$ and $S2$ from a namespace U stored in Bloom Filters $B1$ and $B2$ respectively, then we try to estimate elements that are in $S1$ but not in $S2$.
$SolutionOverview$. We exploit Bloom Sample Tree [SBBR] to represent the entire namespace. We go down the tree estimating the number of common elements between nodes of the tree and $S1$ and $S2$. If no common element with $S1$ is found, search along that path is pruned, if no common element with $S2$ is found, the node is added into the result and if there are common elements with both sets are found, we move a level down on the path.



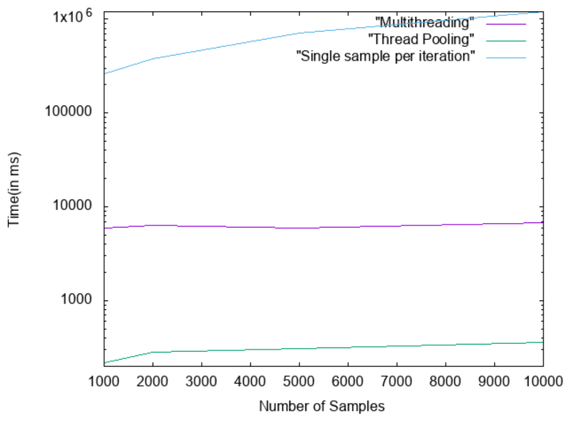
(a) $m = 10^4$, $n = 10^4$, $M = 10^5$



(b) $m = 10^5$, $n = 10^4$, $M = 10^6$



(c) $m = 10^6$, $n = 10^4$, $M = 10^7$



(d) $m = 10^6$, $n = 10^4$, $M = 10^8$

Figure 1: Time taken for different approaches

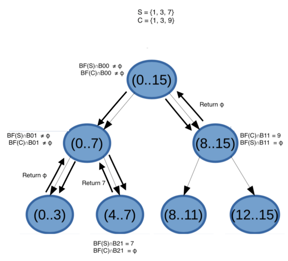


Figure 2: Calculation of set difference

Experimental Analysis We use a synthetic namespace M $(=[1, 2, 3, ...10^6])$. The Bloom Filter size is $50,000$. Both sets A, and B are sampled uniformly at random without replacement from M. Initially, the size of B is fixed to 500, and the size of A is varied in $(10, 20, 40, 80, 160, 320, 640, 1280)$. Similarly, later, the size of A is fixed to 500, and the size of B is varied in the same set.

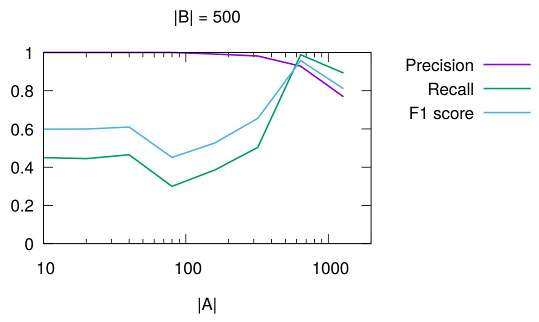
$A - B$ was calculated independently keeping the sizes of one set constant and varying the other. Precision, recall and F1-score were calculated for the experiments **Results** Figure 3 describes precision, recall and F1-score variation in calculating the Set difference between two sets.

Algorithm 1 $Set\_diff(B_{ij}, S, C)$

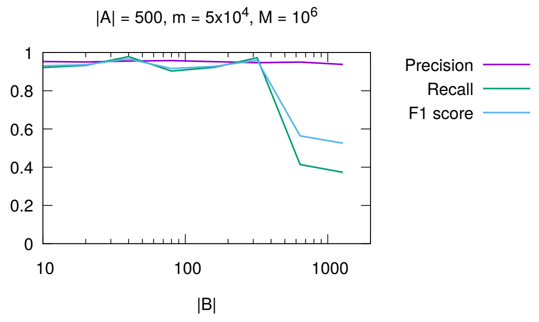
```
1: procedure SET_DIFF($B_{ij}, S, C$)
2:     if $isLeaf(B_{ij})$ then
3:         return all elements of $B_{ij}$ that are in $S$ and not in $C$
4:     else if $B_{ij} \cap S = \phi$ then
5:         return $\phi$
6:     else if $B_{ij} \cap C = \phi$ then
7:         return $S$
8:     else
9:         return $Set\_diff(B_{i+1,2j}, S, C) \cup Set\_diff(B_{i+1,2j+1}, S, C)$
```

Algorithm 2 $Set\_difference(B_{ij}, S, C)$

```
1: procedure SET_DIFFERENCE($B_{ij}, S, C$)
2:     return $Set\_diff(B_{0,0}, S, C)$
```



(a) $A - B$ with varying B and constant A



(b) $A - B$ with varying A and constant B

Figure 3: F1-score for Set difference

3 Detection of Communities in Graphs with Overlapping Communities

Communities are a significant part of many networks like social networks as they tell about relationships between different people in the network. In the real world, it can be seen that a single entity may belong to multiple communities. Thus, it can be an important problem to find communities in graphs with overlapping communities.

One might find it useful use Bloom Filters as a storage data structure because of their space-efficient nature.

Problem Statement. [WHHC12] describes the community detection formally as: Given an undirected network $G = (V, E)$ where V represents the set of vertices in the graph and E the set of edges, we have perfect knowledge of the connectivity of some set of vertices, i.e., the known local community of the graph, which is denoted as C. Necessarily, a set of vertices N that are adjacent to vertices in C but do not belong to C can be partially known (note "partially" means that the complete connectivity of any vertex in N is unknown). Moreover, we define the shell vertex set as region S, where any vertex $v \epsilon S$ has at least one neighbour in N. Let assume that the only way to gain further knowledge about G is to choose one vertex from N and merge it into C. Thus the additional unknown vertices may be added to N. The task of this problem is to constitute a local community C from a single source vertex by this one-vertex-at-a-time step.

Solution Overview. Given an undirected network G = (V, E) where V represents the set of vertices in the graph and E the set of edges. Let us denote shell nodes as set S and community node as set C.Any vertex v ϵ S has at least one neighbour in C and v $\notin$ C.
To detect communities, we are going to expand around a seed node. The algorithm uses a fitness

measure. Fitness Measure value describes how well-connected community is. The algorithm only adds those nodes one-by-one to the community which increases community fitness. We stop when an addition of no node increases the fitness function

Alpha-Measure[LFK]. It is a community fitness measure we used in our algorithm for measuring community fitness.
K_{in} denotes the sum of in-degree and out-degree of the nodes in the community. K_{out} denotes the sum of out-degree of the nodes whose one end of the edge is in the community, and another end is connected with a node outside community.
α is a hyper-parameter which can be adjusted accordingly.

$$F_S = \frac{k_{in}^S}{(k_{in}^S + k_{out}^S)^\alpha}$$

3.1 Algorithm

The algorithm to generate communities from a graph containing communities with highly overlapping communities can be summarized as follows:

1. Initialize shell and community as empty bloom filters.
2. Select a seed, push it and its neighbors into the community and shell bloom filters respectively.
3. Sample some nodes from the shell, and choose the node which results in highest community fitness.
4. Push the node found in above step and push it into the community.
5. Update the shell as set difference of shell and community.
6. Repeat 3-5 until no node is found which increases the community fitness.

The following is the formal algorithm. *seed* refers to initial seed for the procedure, α is the hyper-parameter α to be used for $\alpha\_measure$ and k is the number of elements to be sampled from the *shell* Bloom Filter.

3.2 Experiments and Results

In our experiments, we compare the results of the following algorithms:

1. **LFM**
   This is a basic algorithm described in [LFK]. It also uses the *Alpha Measure.*
2. **PRN**
   This algorithm is described in [ACL08].
3. **GCE**
   This algorithm finds cliques in the graph and finds communities around those cliques as seeds. It is described in [LRMH10].
4. **GCE Local**
   It is a local version of GCE that we implemented so that results can be compared. It can be described as follows. Given seed node s, collect all neighbors of $s = N$. Find cliques in the subgraph induced by N. i.e. the subgraph that has only the nodes in N, and any edges of the graph that are between nodes in N. Expand cliques using the clique expansion algorithm [LRMH10].
5. **1 node expansion**
   We expand community using a seed node. We sample neighbouring nodes which have an edge to the community. We find the fitness function after adding each of the neighbouring nodes individually to the community and add the node to the community which causes the maximum increase in the fitness function. We stop the algorithm when no more nodes can be added to the community which increases fitness function.

Algorithm 3 *Community_detection*$(seed, k, \alpha)$

```
procedure COMMUNITY_DETECTION(seed, k)
    $community \leftarrow seed$                                  ▷ Community Bloom Filter
    $shell \leftarrow adj(seed)$                                 ▷ Shell Bloom Filter
    $size \leftarrow 1$
    $k_{in} \leftarrow 0$
    $k_{out} \leftarrow |adj(seed)|$
    $\alpha\_measure \leftarrow 0$
    while True do
        $candidate\_list \leftarrow sample(shell, k)$     ▷ Sample $k$ elements from the shell Bloom Filter
        $node \leftarrow NULL$
        $save_{in} \leftarrow -1$
        $save_{out} \leftarrow -1$
        for $v \epsilon candidate\_list$ do
            $x_{in} \leftarrow 0$
            for $z \epsilon adj(v)$ do                           ▷ Add comment here
                if $is_i n(community, z)$ then
                    $x_{in} \leftarrow x_{in} + 1$
            $x_{out} \leftarrow |adj(v) - x_{in}|$
            $temp_{in} \leftarrow k_{in} + 2 * x_{in}$
            $temp_{out} \leftarrow k_{out} - x_{in} + x_{out}$
            $\alpha\_measure\_temp \leftarrow$ temp$_{in}/(temp_{in}+temp_{out})^{\alpha}$
            if $\alpha\_measure\_temp > max$ then
                $max \leftarrow \alpha\_measure\_temp$
                $node \leftarrow v$
                $save_{in} \leftarrow temp_{in}, save_{out} \leftarrow temp_{out}$
        if $max > \alpha\_measure$ then
            $\alpha\_measure \leftarrow max$
            $community \leftarrow community \cup node$
            $shell \leftarrow Set\_diff(shell, community)$
            $k_{in} \leftarrow save_{in}, k_{out} \leftarrow save_{out}$
        else
            break

    return $community$
```

6. **Randomized with lists**
   We expand community using a seed node. We sample k nodes which have an edge to the community. We find the fitness function after adding each of the k nodes individually to the community and add the node to the community which causes the maximum increase in the fitness function. We stop the algorithm when no more nodes can be added to the community which increases fitness function.

3.2.1 Datasets

We use an artificially generated undirected graph of $100,000$ nodes for the experiments. The Graph is generated using CKB generator [CKB+14]. Parameters used are mentioned in Figure 4. The communities in the graph are highly overlapping.

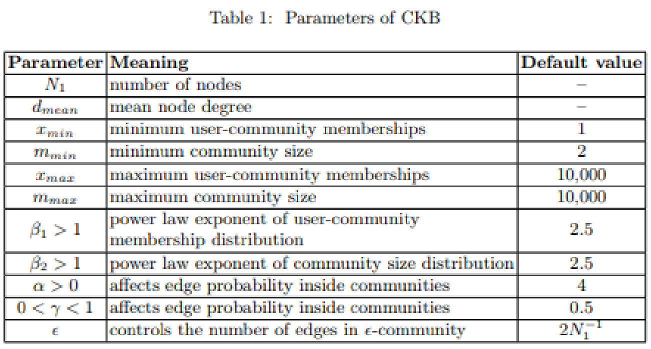


Table 1: Parameters of CKB

Parameter	Meaning	Default value
N_1	number of nodes	–
d_{mean}	mean node degree	–
x_{min}	minimum user-community memberships	1
m_{min}	minimum community size	2
x_{max}	maximum user-community memberships	10,000
m_{max}	maximum community size	10,000
$\beta_1 > 1$	power law exponent of user-community membership distribution	2.5
$\beta_2 > 1$	power law exponent of community size distribution	2.5
$\alpha > 0$	affects edge probability inside communities	4
$0 < \gamma < 1$	affects edge probability inside communities	0.5
ϵ	controls the number of edges in ϵ-community	$2N_1^{-1}$

Figure 4: Parameters used to generate graph

3.2.2 Evaluation

We use F1-score, precision and recall as metrics to evaluate the performance of the algorithms. Precision is the fraction of correctly classified vertices in the community and recall is the ratio of the number of correctly classified vertices to the total number of vertices that should be agglomerated into the community. F1-score id the harmonic mean of precision and recall.

$$F = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

3.2.3 Results

The community is predicted using above algorithms around seed nodes. The predicted community's F-score is calculated with the ground truth communities. The following table displays the results calculated with bloom filter size $= 3500$.

Table 1: Experiment 1 (100,000 nodes)

Algorithm	k	Memory(MB)	Precision	Recall	F-Score	Time(ms)
Page Rank Nibble	-	60.78	0.06	0.16	0.05	35.31
LFM	-	60.78	0.6	0.11	0.05	1,298.02
GCE	-	937.45	0	0.33	0	
GCE Local	-	206.59	0.55	0.59	0.52	385.42
1 Node Expansion	-	1.11	0.58	0.54	0.53	1,416.47
Randomized with Lists	20	1.13	0.67	0.29	0.40	409.57
Randomized with Lists	30	1.12	0.81	0.42	0.5	214.49
Randomized with Lists	40	1.14	0.64	0.44	0.51	854.58
Randomized with Bloom Filter	20	0.89	0.80	0.37	0.50	221,855.68
Randomized with Bloom Filter	30	0.89	0.65	0.40	0.48	265,731.79
Randomized with Bloom Filter	40	0.89	0.80	0.42	0.55	203,679.22

We also studied variation of F1-score, precision recall and community size with change in α. The results are summarized in figures 6 and 7. For these experiments, the same CKB generated graph of $100,000$ was used and the size of Bloom Filter was set to $200,000$.

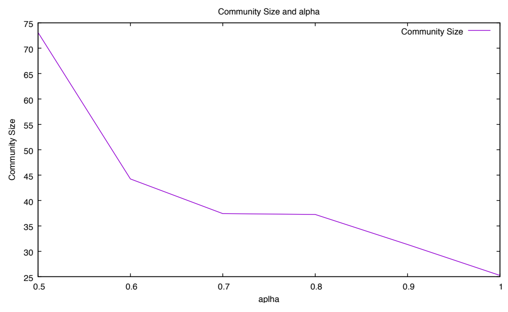


Figure 5: Community Size and α

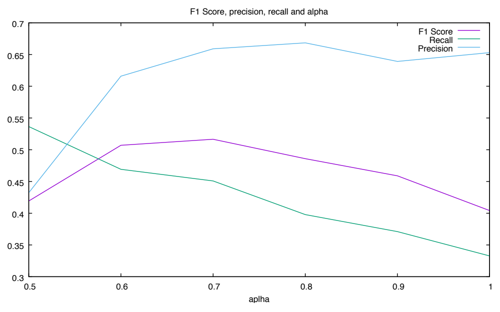


Figure 6: F1 Score, precision, recall and α

References

[ACL08] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. Local partitioning for directed graphs using pagerank. *Internet Mathematics*, pages 3–22,, 2008.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[CKB$^+$14] Kyrylo Chykhradze, Anton Korshunov, Nazar Buzun, Roman Pastukhov, Nikolay Kuzyurin, Denis Turdakov, and Hangkyu Kim. Distributed generation of billion-node social graphs with overlapping community structure. pages 199–208, 2014.

[LFK] Andrea Lancichinetti, Santo Fortunato, and Janos Kertesz. Detecting the overlapping and hierarchical community structure of complex networks.

[LRMH10] Conrad Lee, Fergal Reid, Aaron McDaid, and Neil Hurley. Detecting highly overlapping community structure by greedy clique expansion. 2010.

[PSN10] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *Distributed and Parallel Databases*, 28(2):119–156, Dec 2010.

[SBBR] Neha Sengupta, Amitabha Bagchi, Srikanta Bedathur, and Maya Ramanath. Sampling and reconstruction using bloom filters.

[WHHC12] Ying-Jun Wu, Han Huang, Zhi-Feng Hao, and Feng Chen. Local community detection using link similarity. *Journal of Computer Science and Technology*, 27(6):1261–1268, Nov 2012.