# Sampling Multiple Elements from Bloom Sample Tree and Detecting Communities In Highly Overlapping Graphs using Bloom Filters

Pratham Shah Shubham Jain

November 26, 2023

## 1 Introduction

Bloom Sample Trees [SBBR] can be very effectively used to represent a namespace using Bloom Filters. In this paper, we present solutions of three different problems, with the help of Bloom Sample Trees.

## 2 Sampling Multiple Elements

To sample multiple elements from a set stored in a Bloom Filter, we can extract an element multiple times from a Bloom Filter as explained in [?]. However, this method can be very slow when the namespace and number of elements to be samples is very large. Thus, we look at different approaches to sample multiple elements from a set represented as a Bloom Filter.

### 2.1 Algorithm

We describe different approaches to extract multiple samples from Bloom Sample Tree apart from extracting one sample from a Bloom Sample Tree multiple times.

#### 2.1.1 Approach 1 (Thread Pool)

1. A fixed number of threads are created.
2. To sample n elements, a job queue is created, and the task is added to the job queue.
3. Whenever a thread is idle in the thread pool, it fetches a job from job queue and samples an element independently.

#### 2.1.2

Approach 2 (Thread spawning)

In this approach, we start with the root node, compute the estimated number of elements in the intersection of the query set and both the children, if:

a. Estimates for both children are zero; then we tell the parent node that no elements were found.

b. One estimate shows some elements, then we create a thread for the child node and search along that path.

c. Both the estimates show some elements; we create two threads for the children and sample elements from each path, some elements proportional to the estimated number of elements in the intersection of in both the children respectively.

On reaching a leaf node, the required number of elements are sampled.

### 2.1.3 Approach 3 (Combining Approach 1 and 2)

1. Here we create some threads, equal to the number of nodes in bloom sample tree and if there are more nodes than a threshold, then we use a fixed number of threads.
2. Assign each thread to the number of nodes equal to ratio, where ratio=nodes/threads.
3. Every thread has access to pointers to it's assigned nodes.
4. Every node has a state array which stores number of elements to be sampled at that node.
5. Whenever a node has non-zero state array it will divide the samples proportional the estimated number of intersections of query bloom filter with left and right nodes.
6. Parent thread will update the state array of child nodes according and notify that they have some elements to be sampled.
7. It keeps happening till we reach leaf node where leaf samples number of elements equal to value in its state array and store it in the result array.

## 2.2 Experimental Evaluation

In this section, we describe the experiments we did with different approaches using a static namespace. We compared running time of different approaches to generating a single sample the required number of times.

### 2.2.1 Setup

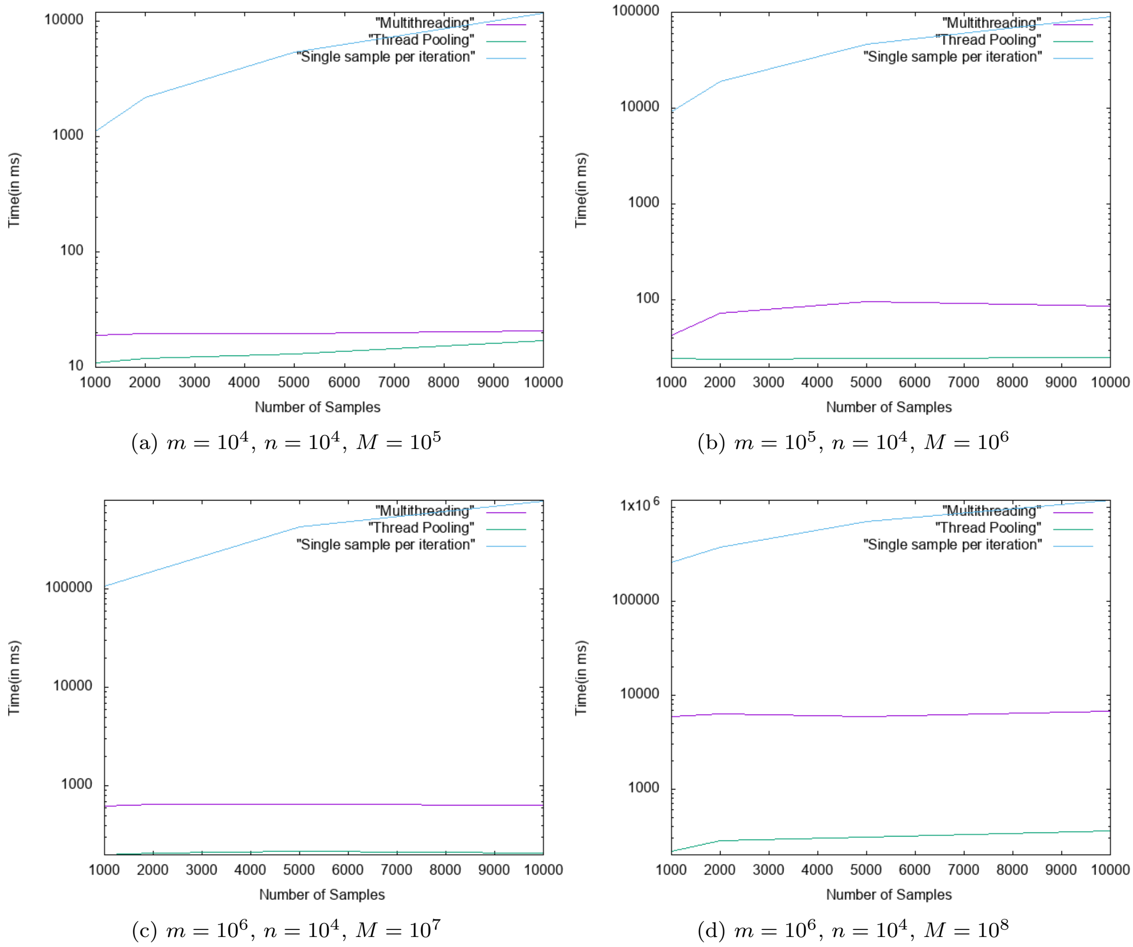
We used a static synthetic namespace to compare the running time of the different approaches. We varied our namespace from $10^5$ to $10^7$. We also varied the number of samples to be generated from 1000 to 10000.In the combined approach, we have used 256 threads for the below experiments.
**Query Sets** In the experiments we performed; we generated query sets uniformly at random from the entire namespace.
**Algorithms** We compared the running time of approaches 2 and 3 and compared it to running time of generating a single sample the required number of times.
**Metrics** We report the running time of algorithms by running them multiple times and taking an average of the running times.

## 2.3 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.

# 3 Calculating Set Difference Among Sets Stored in Bloom Filters

Bloom Filters [BLO70] are data structures that can efficiently answer set-membership queries. Swamidass & Baldi (2007) [SB07] demonstrated how the size of union and intersection of two sets represented using Bloom Filters can be estimated. However, a very fundamental problem of calculating the set difference between two sets stored as Bloom Filters remains solved. In this paper, 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 [?] 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

## 3.1 Algorithm

The following algorithm is used to estimate elements in the set difference among two sets $S$, and $C$ represented as Bloom Filters using Bloom Sample Tree $B$ which represents namespace where $B_{ij}$ represents $j^{th}$ node of $i^{th}$ level.

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

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

## 3.2 Experimental Setup

1. Namespace (M) = $[1, 2, 3, ...10^6]$

2. Bloom filter size (m) = 50000

3. Both sets A and B are sampled uniformly at random without replacement from M.

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

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

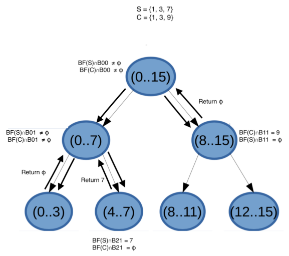


Figure 2: Calculation of set difference

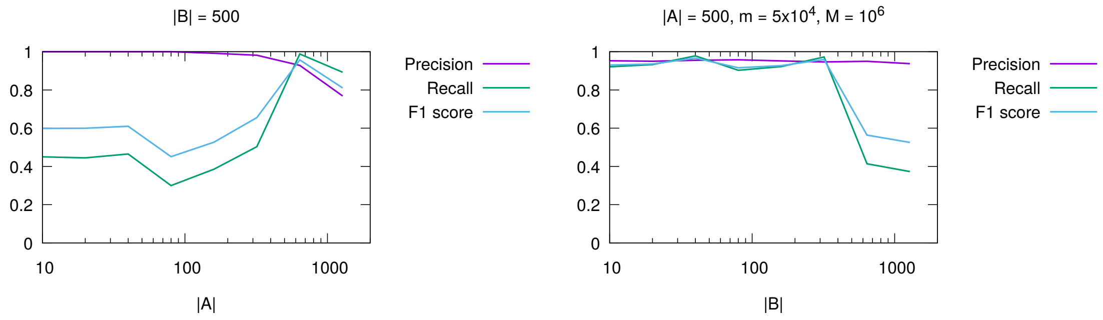
4. In AResults, size of B is fixed to 500, and the size of A is varied in (10,20,40,80,160,320,640,1280).

5. Similarly in BResults, size of A is fixed to 500, and 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

### 3.3 Results

Figure 3 describes precision, recall and F1-score variation in calculating the Set difference between two sets.



(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

## 4 Detection of Communities in Overlapping Graphs

Communities are a significant part of a lot of 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 overlapping graphs.

One might find it useful to save a network as a collection of adjacency lists stored as Bloom Filters as it can be very space efficient. Here, we propose a method to detect communities in overlapping bloom graphs (graphs represented as adjacency lists, stored as Bloom Filters).

*Problem Statement.* Given a Graph $G$ with vertices $V_1, V_2, ..., V_n$ and edges $E_1, E_2, ..., E_m$, find all the communities $C_1, C_2, ..., C_l$ in the graph.
*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 $\epsilon$ 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 to the community which increases community fitness.
*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.
$\alpha$ is a hyper-parameter which can be adjusted accordingly.

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

## 4.1 Algorithm

The following algorithm is used to generate communities from an overlapping Bloom Graph.

**Algorithm 3** *Community_detection(seed)*

```
procedure Community_detection(seed)
    $size \leftarrow 1$
    $b \leftarrow seed$                                   ▷ Community Bloom Filter
    $numCandidates \leftarrow 100$                        ▷ Hyper-Parameter can be changed as per need
    $measure_b \leftarrow 0$                              ▷ conductance or alpha-measure
    while True do
        $candidate\_list \leftarrow \phi$                 ▷ This is a list of tuples
        for (j = 1 to min(numCandidates, size) do
            sample element u from Bloom filter b
            sample element v from adj(u)
            if v in b then
                continue
            $t \leftarrow$ numOnes in $adj(v) \cap b$
            $candidateList \leftarrow candidatelist.append((v, t))$   ▷ append tuple (v,t) to the list
            sort candidateList by the second element in each tuple
            truncate candidateList to top K
            put candidateList in another Bloom filter C
            if inserting c into b increases measure_b then
                $b \leftarrow b \cup c$
                $size \leftarrow size + K$
                $updatemeasure\_b$
            else
                break
    return b
```

## 4.2 Experimental Setup

1. Graph is generated using CKB generator [CKB+14]. Parameters used are as mentioned in Figure 4.

2. Graph is undirected and communities are highly overlapping.

3. Number of Nodes in Graph = 100,000 and 1,000,000.

4. F-score of a community predicted using the above algorithm are calculated with the communities in ground truth.

5. We have used alpha = 0.75 (hyper-parameter) and m = 2*number of nodes.

6. The following algorithms were tested:

   (a) **Page Rank Nibble [ACL08]**

   (b) **LFM [LFK]**

   (c) **GCE [LRMH]**

   (d) **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.

   (e) **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.

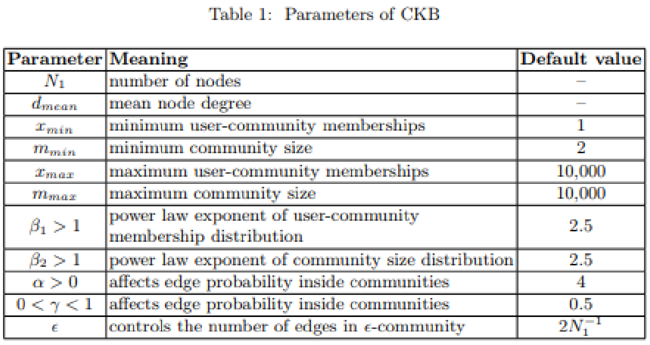


Table 1: Parameters of CKB

| 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 |
| $\epsilon$ | controls the number of edges in $\epsilon$-community | $2N_1^{-1}$ |

Figure 4: Parameters used to generate graph

## 4.3 Results

1. Community is predicted using above algorithm around a seed node.

2. The predicted community's F-score is calculated with the ground truth communities.

3. Below results are collected as an average of 10 experiments.

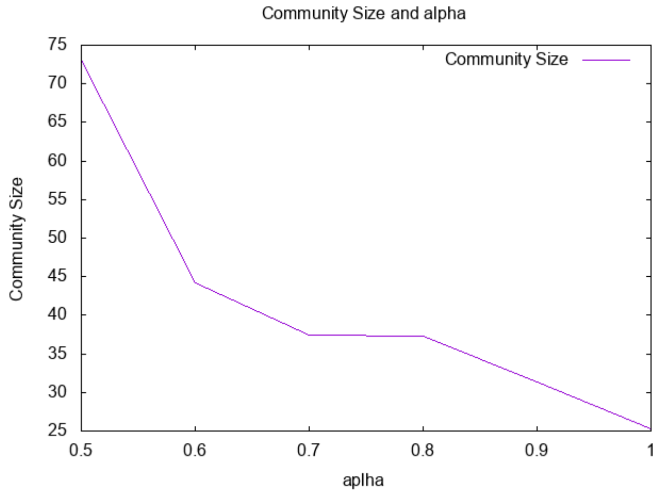
Table 1: Experiment 1 (100,000 nodes)

| 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 | |
| 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 |

Table 2: Experiment 2 (1,000,000 nodes)

| Algorithm | k | Memory(MB) | Precision | Recall | F-Score | Time(ms) |
|---|---|---|---|---|---|---|
| GCE | | | | | | |
| 1 Node Expansion | - | 11.16 | 0.51 | 0.47 | 0.48 | 290.30 |
| Randomized with Lists | 20 | 11.17 | 0.53 | 0.29 | 0.34 | 407.20 |
| Randomized with Lists | 30 | 11.18 | 0.58 | 0.48 | 0.52 | 857.14 |
| Randomized with Lists | 40 | 11.18 | 0.76 | 0.49 | 0.58 | 526.77 |
| Randomized with Bloom Filter | 20 | 8.83 | 0.71 | 0.38 | 0.48 | 753,793.62 |
| Randomized with Bloom Filter | 30 | 8.83 | 0.75 | 0.50 | 0.59 | 588,873.63 |
| Randomized with Bloom Filter | 40 | 8.83 | 0.67 | 0.45 | 0.53 | 1,531,642.51 |

4. The graph below represents how community size varies with $\alpha$ (with number of samples: 40, number of nodes: 100K and size of bloom filter: 200K).



Size.png

Figure 5: Community Size and $\alpha$

5. The graph below represents how F1 score, precision and recall vary with $\alpha$ (with number of samples: 40, number of nodes: 100K and size of Bloom Filter: 200K).

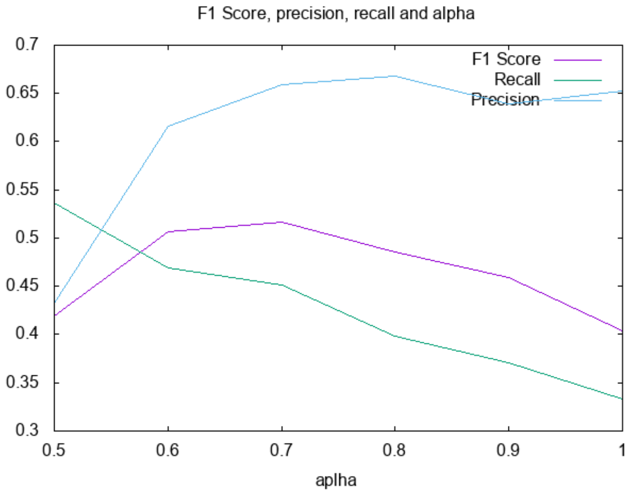


Figure 6: F1 Score, precision, recall and $\alpha$

6. The graph below represents how F1 score varies with $\alpha$ and number of samples taken from Bloom Filter (with number of samples: 40, number of nodes: 100K and size of bloom filter: 200K).

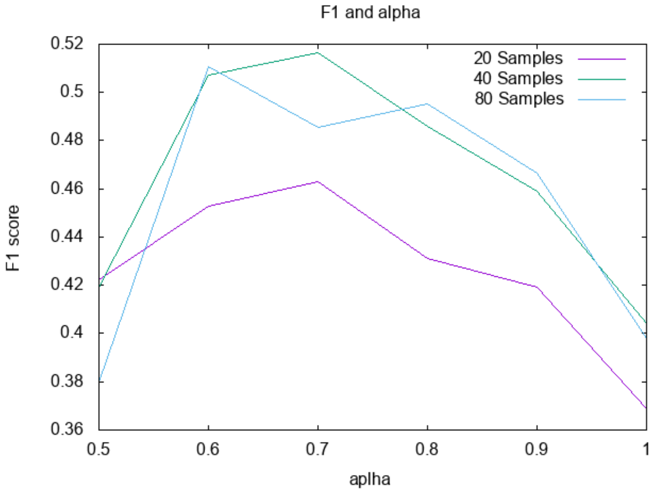


Figure 7

# 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, vol. 13, no. 7*, pages 422–426, 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.

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

[SB07] S. Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling, ACS Publications*, page 952–964, 2007.

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