Decomposing One Signal into Two:
Counting New Neighbors in a Graph with Bloom Filters

David Rodriguez
dvidrdgz@gmail.com

Abstract

Given a sequence of graphs $G_1, G_2, \ldots$ from time $t_1, t_2, \ldots$ we want to flag all newly seen neighbors to a vertex v_i (and also count how much one repeats). It would be nice to be able to store a lot of vertices adjacent to v_i over a long period of time, but that might be infeasible. Hence, we approximate the set of neighbors within a time window using a bloom filter. We can *union* multiple bloom filters filled with neighbors to achieve our desired time window (possibly big) while controlling the memory required. The result is *new* and *repeat* counts, per vertex, forming two time-series. On these time-series we can model the underlying process to identify anomalies and measure the correlation between the two. We provide experimental results on a Houston, Texas government email dataset forming a graph with nodes and edges made between *senders* and *recipients*. Experiments suggest that 80% of network traffic to popular email domains is repeat visitors and thus only 20% are new.

1 Introduction

In this paper we explore efficient methods for sketching historical events using bloom filters. We apply this technique in graph problems where maintaining a sketch of a vertices neighbors over time is difficult. In particular, we want to be able to control how much history to store and cache (with varying size time windows) as well as the control the computational burden required to surface counts related to the number of *new* and *repeat* neighbors a vertex is connected to in a new time window.

The motivation behind these goals, are twofold. First, we wish to develop scalable methods to monitor behavior between computers within a network on premise and in the cloud. In particular, if one manages a cloud infrastructure, it is common for machines to be rotated in and out of commission, posing challenges for modeling and detecting anomalies. Second, as a graph increases the abilithy to efficiently store and maintain topological statistics is desired. And while many spectral graph theoretic techniques are known there are challenges to reproducing an adjancency matrix on time dependent graphs.

In this work we explore approximating a vertices neighbors in bloom filters over different time windows. This technique of buffering bloom filters, then merging them, enables dynamic time windows and aging out old neighbor sets. In addition, this technique enables the cache of neighbor sets to be constructed from novel configurations of windows. For example, one might construct a cache of neighbors for days of the week, month, or hours of the day.

Lastly, and most importantly, this method lays the foundation for a series of streaming algorithms to model the underlying distribution to the *counts* that are discussed herein. Viewing these counts as random variables enables a series of stochastic modeling techniques to be performed.

2 Graph Topologies over Time

A fundamental challenge when modelling graphs over time is the appearance and disappearance of vertices over time. In particular, it is challenging to represent a graph in an adjancency matrix over time, because the row and columns become degenerate: i.e. with no incident edges.

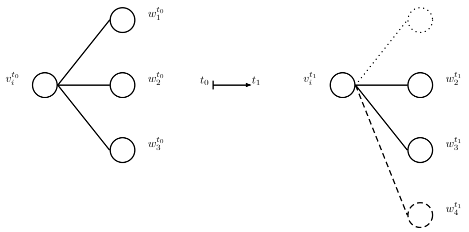

Figure 1: Vertex v_i with different neighbors at t_0 and t_1: w_1 disappears and w_4 appears at t_1.

Another way to model a graph is through adjacency lists. In fact, the adjancency list provides an intuitive way to keep track of nodes connected with directed edges. Figure 2 shows a vertex v_i and neighbors over multiple time periods.

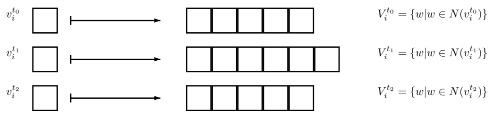
$v_i^{t_0}$
$v_i^{t_1}$
$v_i^{t_2}$
$V_i^{t_0} = \{w | w \in N(v_i^{t_0})\}$
$V_i^{t_1} = \{w | w \in N(v_i^{t_1})\}$
$V_i^{t_2} = \{w | w \in N(v_i^{t_2})\}$

Figure 2: Vertex v_i and neighbors in the set V_i for each time: t_0, t_1, t_2.

There are some nice properties of sets. We can combine the sets of neighbors from different times to form another set. For example, let $V_i^{t_0}, V_i^{t_1}$ be the set of neighbors for a node v_i at at time windows t_0, t_1, respectively. Then,

$$V_i^{t_0} \cup V_i^{t_1} = V_i^{t_0+t_1} \tag{1}$$

is short-hand for the set of all neighbors to v_i between times $[t_0, t_1]$. Figure 3 shows more than two sets can be combined to form a new one.

$$\begin{tikzcd}
v_i^{t_0} \arrow[r, mapsto] & V_i^{t_0} & & & \\
v_i^{t_1} \arrow[r, mapsto] & V_i^{t_1} \arrow[r, mapsto, "f"] & v_i^{t_2} \arrow[r, mapsto] & V_i^{t_0+t_1+t_2} \\
v_i^{t_2} \arrow[r, mapsto] & V_i^{t_2} & & &
\end{tikzcd}$$

Figure 3: Example of merging more than two sets of neighbors to v_i.

Sets provide an easy way to keep track of neighbors that appear and disappear over time. In the next section we discuss ways to measure it.

3 Time Window Lags

One way we can take advantage of the sets of neighbor nodes can be visualized on a timeline. Figure 5 shows the neighbors of a vertex v_i over time represented by color boxes on the upper part of the timeline. On the lower part of the time, we see the upper windows combined into one set holding the distinct elements from the window lag period of three windows.

Now, consider the next time window t and the set of neighbors to v_i. We may wish to compare V_i^t to any other neighbor set from a different time window. For example, we can compare any $V_i^{t_i}$ to $V_i^{t_j}$ from different time windows t_i, t_j. We might compare sets using *union, intersection, difference* operations.

As step further, you might have guessed, is to compare V_i^t to $V_i^{t_0+t_1+t_2}$. Two of the simplest statistics we can compute from this are:

$$V_i^t \setminus V_i^{t_0+t_1+t_2} \tag{2}$$

which provides the number of new neighbors to v_i at time t not in any t_0, t_1, t_2. Or perhaps,

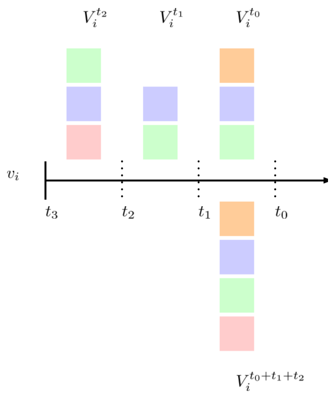

Figure 4: Vertex v_i and neighbors in the set V_i for each time: t_0, t_1, t_2.

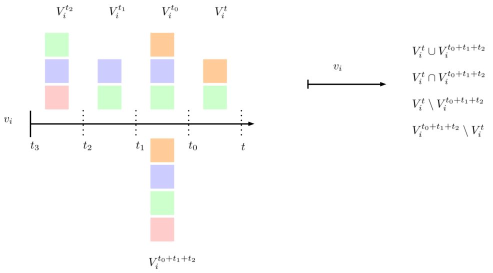
$V_i^{t_2}$
$V_i^{t_1}$
$V_i^{t_0}$
V_i^t
v_i
t_3
t_2
t_1
t_0
t
$V_i^{t_0+t_1+t_2}$
v_i
$V_i^t \cup V_i^{t_0+t_1+t_2}$
$V_i^t \cap V_i^{t_0+t_1+t_2}$
$V_i^t \setminus V_i^{t_0+t_1+t_2}$
$V_i^{t_0+t_1+t_2} \setminus V_i^t$

Figure 5: Example showing how V_i^t can be compared to $V_i^{t_0+t_1+t_2}$

$$V_i^t \cap V_i^{t_0+t_1+t_2} \tag{3}$$

which compates the number of neighbors repeating to node v_i at t an in at least one of t_0, t_1, t_2.

The percent of the newest neighbors repeating might also be of interest. Due to the possible imbalance of comparing a neighbor set from one time window to another neighbor, because the set comprised of multiple windows may contain a lot more elements, a slightly alternate version of the *Jaccard* computation can be performed:

$$\hat{J}(A, B) = \frac{|A \cap B|}{\min\{|A|, |B|\}} \tag{4}$$

4 Approximate Sets via Bloomfilters

A *bloom filter* is a way to maintain a set *approximately*. The trade-off is that for a fixed amount of memory one can keep track of some n items with a certain error δ. We will see that bloom filters act as typical sets (at least *approximately*): i.e. we can take their union and intersection.

Let $BF = \{0, 1\}^K$ represent a bloom fiter: i.e. a bit array of K-dimension. We then associate k uniform hash functions $h_1, \cdots, h_k$ with this bloom filter. A hash function is a map from a set to the bit-space: $h : A \to \{0, 1\}^K$

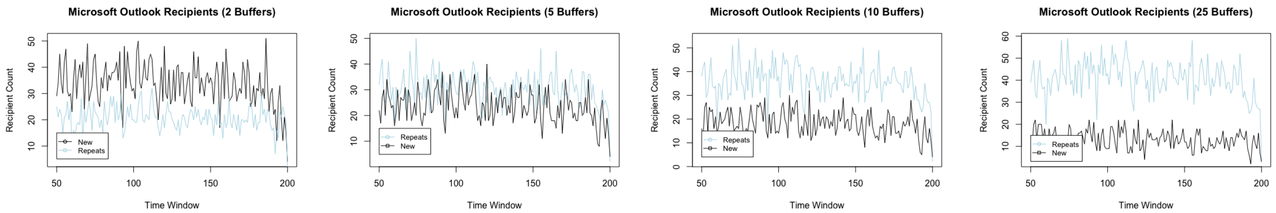

Figure 6: Comparing 2,5,10,25 buffer sizes centered around *Microsoft Outlook* sending mail to other recipients.

Let $BF_i = [b_1, \dots, b_K]$ be a bloom filter and BF_j be defined similarly with variables $\hat{b}_i$. Then, we can take the *union* of two bloom filters

$$BF_i \cup BF_j = [b_1 || \hat{b}_1, \cdots, b_K || \hat{b}_K] \tag{5}$$

where $b_i || \hat{b}_i$ is bit-wise or. We can can take the *intersection* in a similar way by using bit-wise and.

Suppose we want to store 100 elements in one bloom filter with approximately .001 chance of a false-positive (using 15 hash functions). Then we would need the solution to the following equation:

$$.001 = (1 - e^{1500/x})^{15} \tag{6}$$

giving the solution $x \approx 1504.75$. Thus, we need to store 1505 bits. This is an improvement, assuming a 10 character string requires 40 bits and therefore 100 strings require 4000 bits.

Additionally, bloom filters support the *is in* operation. That is, we can check if x is in BF for any bloom filter. The time required to perform the check depends on the number of hashing functions.

5 Caching Bloom Filters and FIFO

Assuming we have a neighbor set V_i^t for a vertext v_i we now just need to define how to age out old buffered windows of neighbor sets. Let $\mathscr{B}_i = \{(v_1^{t_i}, V_1^{t_i}), \dots, (v_n^{t_i}, V_n^{t_i})\}$ be a buffer of vertices and their neighbor sets. We define a q-buffer with $q = \{1, 2, \dots\}$ to be:

$$\mathscr{B} = [\mathscr{B}_1, \dots, \mathscr{B}_q] \tag{7}$$

which holds q discrete time windows of vertices and neighbors. This is convenient, insofar as, we can now age out old buffers and add new ones using *pop/push* operations.

Similarly, we define a *cache*, $\mathscr{C}$, from buffers to be the unique vertices and neighbors within the buffer:

$$\mathscr{C} = [(v_1, V_1^{t_1 + \cdots + t_q}), \dots, (v_n, V_n^{t_1 + \cdots + t_q})] \tag{8}$$

It is worth noting, the computation time required to achieve any $V_i^{t_1 + \cdots + t_q}$ may vary. But, depending on the time windows, this operation may only need to be done once an hour, day, and others.

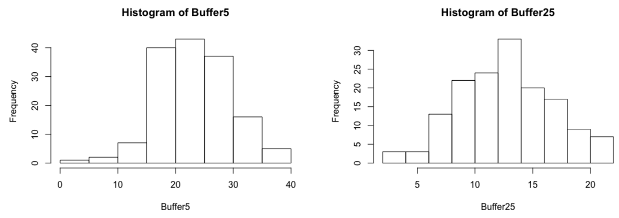

Figure 7: Comparing *new* distribution of counts from buffers 5 and 25.

6 Measuring New/Repeats in Email Graphs

We experiment with the Houston, Texas government email dataset. [1] This data set is over 6M emails with information captured in columns: *Sender, To, CC, BCC, Sent date/time, Received date/time* This email graph is similar to a network graph of computers interconnected and communicating.

To simplify the study we construct directed edges of (sender, to) where we typically call *to* the recipient. A few types of studies are possible: a) vary the buffer size, or b) vary the window sizes. To start, we fixed the window sizes over 1M emails, breaking the emails into 200 windows. Each window, therefore, had $1M/200$ emails. In the first experiment, we increased the the buffer size from 2, 5, 10, 25.

Observation This first experiments tested the reduction in *new* and *repeat* recipient counts. The intuition, was the larger the buffer the more *repeat* counts, with fewer *new*. Figure 6 compares side-by-side the buffer sizes of 2,5,10,25. It can be seen that increasoing the buffer size does increase the number of *repeat* counts.

One interesting question is whether the distribution of the counts in the *new* changes when we change the buffer size. Figure 7 compares the distribution of counts from the 5 and 25 buffer experiments.

7 Conclusion

In this paper we begin to explore ways of maintaining a history of neighbors to a vertex over time. We then observed

[1] https://data.world/sketchcity/city-of-houston-email-metadata-january-march-2017

on an email graph, how we can alter the sender *Microsoft Outlook*’s number of recipients in each time window by the buffer sizes 2, 5, 10, 25.

Next, steps are to model and measure the time-series of counts and use an online algorithm to detect anomalies.

A Source Code

```
from datetime import datetime
from pybloom import BloomFilter
import random


def parse_datetime(s):
    """ parse datetime from file """
    return datetime.strptime(s, "%Y-%m-%d %H:%M:%SZ")


def random_cuts(n, k):
    """ need random break points for a list """
    return sorted([random.randint(0, n) for i in range(k)])


def even_cuts(n, k):
    """ need random break points for a list """
    return range(0, n, n / k)[1:]


def emitter(l, n):
    """
    emit sublists of l, in a total of n batches
    """
    ln = len(l)
    ks = [0] + even_cuts(ln, n) + [ln]

    for i, j in zip(ks, ks[1:]):
        yield l[i:j]


class Row(object):
    """
    Simple class to parse each row line
    we can capture (or ignore) malformed lines
    if we just add a try/catch
    """

    def __init__(self, row):
        parts = row.strip()[1:-1].split(’\",\"’)
        self.sender = parts[0]
        self.recipient = parts[1]
        self.time = parse_datetime(parts[4])


    def get_sender(self):
        return self.sender

    def get_recipient(self):
        return self.recipient

    def get_time(self):
        return self.time

    def get_hour(self):
        return self.time.hour

    def get_minute(self):
        return self.time.minute


def group_by_sender(emails):
    sender_groups = {}
    """ given a list of email objects return sender, recipients dictionary """
    for email in emails:
        sender = email.sender
        recipient = email.recipient

        if sender in sender_groups:
            sender_groups[sender].add(recipient)
        else:
            sender_groups[sender] = set([recipient])

    return sender_groups


def bloomify(senders):
    """
    given sender, recipient dictionary
    return sender, bloom filter filled with recipients
    """
    tmp = {}
    for k, vs in senders.items():
        bf = BloomFilter(capacity=20000, error_rate=0.001)
        for v in vs:
            _ = bf.add(v)
        tmp[k] = bf
    return tmp


def flatten(l):
    """
    Simple algorithm to flatten a list of dictionaries
    where the keys are strings, values bloomfilters
    """
    tmp = {}
    for b in l:
        for k, v in b.items():
            if k in tmp:
                old_bf = tmp[k]
                new_bf = old_bf.union(v)
                tmp[k] = new_bf
            else:
                tmp[k] = v
    return tmp


class Buffer(object):
    buffer = []
    cache = {}

    def __init__(self, count):
        self.count = count

    def add_buffer(self, buf):
        if len(self.buffer) > self.count - 1:
            self.buffer.pop()
        self.buffer = [buf] + self.buffer

        tmp = flatten(self.buffer)
        self.cache = tmp

    def is_key_in_cache(self, k):
        return k in self.cache

    def split(self, k, vs):
        """ return the number of new, and old recipients """
        new_vs = 0
        intersection = 0
        for v in vs:
            if v in self.cache[k]:
                intersection += 1

            else:
                new_vs += 1
        return intersection, new_vs

    def intersection(self, k, v):
        """ return intersection of two bloom filters """
        return self.cache[k].intersection(v)

    def union(self, k, v):
        """ return union of two bloom filters """
        return self.cache[k].union(v)


def write(writer):
    with open("outfile_1m_buf25_split200.csv", "a") as f:
        for w in writer:
            strify = map(str, w)
            f.write(",".join(strify) + "\n")


if __name__ == "__main__":
    path = "coh_email_metadata_1Q17.csv"

    emails = []
    with open(path, ’r’) as f:
        for line in f.readlines()[1:1000000]:
            try:
                emails.append(Row(line))
            except:
                print "[ERROR] parsing: {}".format(line)

    emails = sorted(emails, key=lambda x: (x.get_hour(), x.get_minute()))

    buffer = Buffer(25)
    group = 0
    for window in emitter(emails, 200):
        group += 1
        senders = group_by_sender(window)

        writer = []
        for sender, recipients in senders.items():
            if buffer.is_key_in_cache(sender):
                repeat_recips, new_recips = buffer.split(sender, recipients)
                writer.append((group, sender, repeat_recips, new_recips))
            else:
                writer.append((group, sender, 0, len(recipients)))

        write(writer)

        senders_bf = bloomify(senders)
        buffer.add_buffer(senders_bf)
```