CopyCatch: Stopping Group Attacks by Spotting Lockstep Behavior in Social Networks

Alex Beutel*
Carnegie Mellon University
Pittsburgh, PA
abeutel@cs.cmu.edu

Wanhong Xu
Facebook
Menlo Park, CA
xu@fb.com

Venkatesan Guruswami
Carnegie Mellon University
Pittsburgh, PA
guruswami@cmu.edu

Christopher Palow
Facebook
London, UK
cpalow@fb.com

Christos Faloutsos
Carnegie Mellon University
Pittsburgh, PA
christos@cs.cmu.edu

ABSTRACT

How can web services that depend on user generated content discern fraudulent input by spammers from legitimate input? In this paper we focus on the social network Facebook and the problem of discerning ill-gotten Page Likes, made by spammers hoping to turn a profit, from legitimate Page Likes. Our method, which we refer to as CopyCatch, detects lockstep Page Like patterns on Facebook by analyzing only the social graph between users and Pages and the times at which the edges in the graph (the Likes) were created. We offer the following contributions: (1) We give a novel problem formulation, with a simple concrete definition of suspicious behavior in terms of graph structure and edge constraints. (2) We offer two algorithms to find such suspicious lockstep behavior - one provably-convergent iterative algorithm and one approximate, scalable MapReduce implementation. (3) We show that our method severely limits "greedy attacks" and analyze the bounds from the application of the Zarankiewicz problem to our setting. Finally, we demonstrate and discuss the effectiveness of CopyCatch at Facebook and on synthetic data, as well as potential extensions to anomaly detection problems in other domains. CopyCatch is actively in use at Facebook, searching for attacks on Facebook's social graph of over a *billion* users, many *millions* of Pages, and *billions* of Page Likes.

Categories and Subject Descriptors

I.5.3. [**Computing Methodologies**]: Pattern Recognition—*Clustering*

Keywords

Anomaly detection; Social networks; Bipartite cores

1. INTRODUCTION

When on the web, how can we trust content generated by other users? As the web has become an increasingly integral part of our daily lives, from work to shopping to socializing, it has become a focus of spammers attempting to make money off Internet users, even if it takes dubious means.

In recent years, web services have increasingly relied on social data to provide information to their users. For example, Facebook users discover content based on what their friends and other users like, and on Amazon users evaluate potential purchases based on other users' reviews. Unfortunately, attackers attempt to skew content perception by offering misleading feedback (through a variety of means), with the goal of increased distribution for their content. The challenge becomes distinguishing such "fake" feedback from legitimate user feedback. This is a challenge for *all* services that depend on user behavior for their algorithms and recommendations, from stories on Facebook to products on Amazon or reviews of businesses on TripAdvisor.

On Facebook, Pages are used by organizations to interact with their fans. Users can "Like" a Page to let their friends know about their interests and to receive content from that Page in their News Feed, the primary distribution channel on Facebook. Other users may interpret a high Like count as a Page being popular and also will see their friends' Page Likes in their News Feeds.

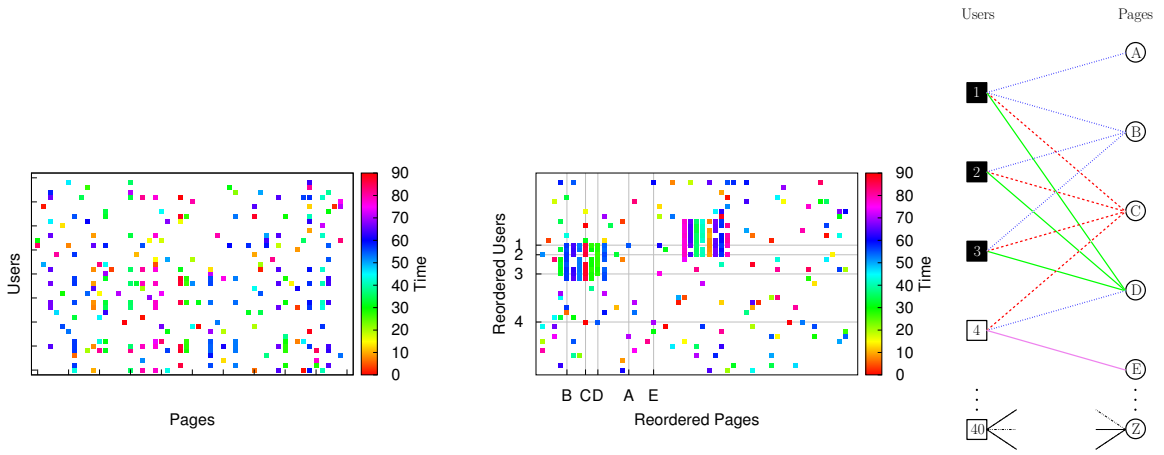
Because of its utility as a distribution channel, attackers frequently attempt to boost Page Like counts to get increased distribution for their content. At Facebook, we have found that attackers have attempted to inflate Like counts through a variety of deceitful methods, including malware, credential stealing, social engineering, and fake accounts. We define an ill-gotten Like, including those from the methods named previously, as "a Like that doesn't come from someone truly interested in connecting with a Page" [12]. As Facebook Security recently posted:

> Real identity, for both users and brands on Facebook, is important to not only Facebook's mission of helping the world share, but also the need for people and customers to authentically connect to the Pages they care about. When a Page and fan connect on Facebook, we want to ensure that connection involves a real person interested in hearing from a specific Page and engaging with that brand's content. [12]

In this paper we focus on the problem of detecting ill-gotten Page Likes on Facebook, and describe how our algorithm can

*A portion of this work was done while working at Facebook.

WWW 2013, May 13–17, 2013, Rio de Janeiro, Brazil.
ACM 978-1-4503-2035-1/13/05.



(a) Without CopyCatch (b) With CopyCatch (c) Graphical view

Figure 1: A toy example of Page Likes over time with a subset of users and Pages organized to clearly show two detected attempts to inflate Page Like counts.

also be used for detecting deceitful user feedback in many other online settings such as product reviews. For all of these problems the mission Facebook describes above holds - user generated content must be honest and legitimate if users are to trust and get value out of that web service.

In solving this problem, we attack the spammers at their critical weakness. For spammers to be successful they need *many* users to Like their Pages. However, Facebook already has many anti-phishing [11] and anti-malware [10, 9] mechanisms making it difficult for real accounts to be compromised, and many algorithms to detect fake accounts [25]. As a result, it is hard for an adversary to control many accounts, and instead they need to use the same few to Like many Pages. Therefore, we look for *lockstep behavior* - groups of users acting together, generally Liking the same Pages at around the same time. We call our algorithm to detect such behavior CopyCatch and describe a process of *multi-user rate limiting* where, because of our constraint on Like times, we limit the rate at which a group of users can perform actions together (much stricter than you can limit individual users). Figure 1 demonstrates the challenge and the strength of CopyCatch in detecting such behavior.

To detect attackers attempting to deceive users, we take a graph based approach to the problem. As we see in Figure 1(c), in the case of Facebook Page Likes, we have a bipartite graph between users and Pages, with the time at which each edge (Page Like) was created. Our algorithm searches for near-bipartite cores, where the same set of users Like the same set of Pages, and add constraints on the relationship between the edge properties (Like times) in this core. This can be extended to the bipartite graph of users to products where edges represent product reviews, or to general graphs such as the user to user connections of Instagram followers.

Our paper offers a number of contributions, which build toward solving this problem:

1. **Problem Formulation:** We offer a novel problem formulation to a relevant, real-world challenge realized at Facebook and relevant in many online settings. We call our approach CopyCatch.
2. **Algorithm:** Pulling from work in one-class clustering and subspace clustering, we offer two algorithms to spot lockstep behavior: a provably-convergent serial iterative algorithm, and an approximate, scalable MapReduce implementation.
3. **Theoretical Analysis:** We show that catching anomalous behavior, as we have defined and as our algorithms detect, severely limits the damage an adversary can do when following a "greedy attack" strategy. We then apply research on the Zarankiewicz problem to our setting, showing that it is hard to find an optimal strategy against CopyCatch.

In Section 2 we describe related work in local clustering, subspace clustering, MapReduce, and anomaly detection. In Section 3 we give our problem formulation. In Section 4 we formulate the problem as an optimization problem, describe our serial algorithm, and prove that it converges. In Section 5 we describe our MapReduce algorithm and implementation. Finally in Section 6 we discuss the worst case damage an adversary could inflict, in Section 7 we offer experiments demonstrating the usefulness of our implementation at Facebook and on synthetic data, and in Section 8 we discuss the applicability of our approach to problems in other domains.

2. RELATED WORK

Our work pulls from many different fields of research. We describe a few below and mention others later in the text.

2.1 Local clustering

Clustering is one of the classic problems in both machine learning and data mining, with a wide range of methods still being developed. In this research, we build off Crammer et al.'s work on local one-class optimization and related work [6, 15], which focuses on finding dense clusters in noisy data through local search. Our algorithm also operates similarly to mean-shift clustering with a flat kernel [5].

2.2 Co-clustering and subspace clustering

For our problem we aim to find sets of users Liking the same set of Pages at around the same time. Given a data matrix of users $\times$ Pages, this requires clustering both the rows and columns of the matrix. The problem of partitioning both the rows and columns of a matrix is known as co-clustering or bi-clustering. The problem is NP-hard, so many approaches use an approximation of the problem.

There has been extensive research on co-clustering including [3, 4, 8, 21], with applications ranging from collaborative filtering [14] to anomaly and intrusion detection [20]. Papadimitriou et al. [19] offer an iterative distributed algorithm for performing co-clustering with MapReduce.

Similarly, there has been much work on the general problem of clustering high dimensional data. Within this work, subspace clustering focuses on finding subsets of features that are relevant for clustering a subset of the data. [16] gives a good comprehensive survey of the recent research in clustering high-dimensional data.

2.3 MapReduce

To make our algorithm scale to large, web-scale data, we implement our algorithm in the MapReduce framework [7]. Hadoop [1] is an open source implementation of the MapReduce framework that is widely used. Facebook has a large Hadoop installation on which we built our implementation. In general, Hadoop and the Hadoop file system (HDFS) offer a distributed platform to store data and run parallel algorithms over a cluster of computers. We will give more details about the data flow and capabilities of Hadoop in Section 5.

2.4 Graph-Based Anomaly Detection

There has been extensive work in anomaly detection on web data, and much of it has focused on using graphs for spotting anomalies. For example, [17, 23] focus on finding novel subgraphs in large networks. In this work we focus on finding near bipartite cores with certain edge constraints. [18] uses belief propagation to find near bipartite cores on the eBay graph, and [24] takes an SVD-like approach to find similar interesting patterns in phone call data. Our problem formulation differs from such prior work in our novel use of edge constraints to discern normal behavior from suspicious behavior.

3. PROBLEM FORMULATION

We now describe the mathematical details of our problem. Table 1 gives a list of the different symbols we will use throughout the paper. In general scalars will be denoted by italic letters e.g. N or ρ, vectors will be denoted by lowercase boldface characters e.g. $\mathbf{c}$, matrices will be denoted by uppercase boldface characters e.g. $\mathbf{L}$, and sets will be denoted by script characters e.g. $\mathcal{P}$. We use subscripts to index into vectors and matrices, where $\mathbf{c}_j$ is the scalar in the jth position of $\mathbf{c}$, $\mathbf{L}_{i,j}$ is the scalar in the ith row and jth column of $\mathbf{L}$, and $\mathbf{L}_{i,*}$ is a vector of the ith row of $\mathbf{L}$.

As we previously described, Facebook has the challenge of preventing adversaries from artificially inflating a Page's "Like count" in an effort to try to improve the Page's legitimacy and get distribution through the site. Since each user can only Like each Page once, a Page's Like count can only be increased through many users Liking the same Page.

Unfortunately, there is no ground truth to whether any individual Page Like is legitimate or not. Therefore, we take an unsupervised approach and only define suspicious behavior in terms of graph structure and edge creation times. Although our methods can be easily extended to many other settings, we will often describe the work in terms of Facebook users, Pages, and Likes for simplicity and clarity.

Before defining suspicious lockstep activity, we must give some notation surrounding our problem. We assume we have a set of users indexed from 1 to N, $\mathcal{U} = \{i\}_{i=1}^{N}$ and a set of M pages $\mathcal{P} = \{j\}_{j=1}^{M}$ similarly indexed. We define $\mathcal{E}$ as the set of edges in the graph, where $(i,j) \in \mathcal{E}$ is user i has Liked Page j. We also define an indicator matrix $\mathbf{I}$ such that $\mathbf{I}_{i,j} = 1$ if $(i,j) \in \mathcal{E}$, and $\mathbf{I}_{i,j} = 0$ otherwise. Last, we define our data matrix $\mathbf{L}$ such that $\mathbf{L}_{i,j} = t_{i,j}$ for all $(i,j) \in \mathcal{E}$, where $t_{i,j}$ is the time at which user i Liked Page j.

Table 1: Symbols and Definitions

Symbol	Definition and Description
N and M	Number of nodes on either side of the bipartite graph (users and Pages)
$\mathbf{L}$	$N \times M$ data matrix of edge data (Like times)
$\mathbf{I}$	$N \times M$ adjacency matrix
$\mathcal{U}$ and $\mathcal{P}$	Set of indices of rows and columns (indexed users and Pages)
n and m	Number of nodes necessary to be considered suspicious for each side of the bipartite core
$\mathcal{P}'$	Subset of columns (Pages) that are suspicious
$\mathbf{c}$	Vector of times for each column around which there are suspicious rows (users)
$2\Delta t$	Width of time window
ρ	Percent of $\mathcal{P}'$ for which an suspicious user must be within the time window
ϕ	Thresholding function to compare two data points
s	Number of clusters being search for in parallel
$\mathcal{P}$	Set of $\mathcal{P}'$ for multiple clusters
$\mathcal{C}$	Set of $\mathbf{c}$ for multiple clusters

We can now broadly define our problem as:

Given: A graph of Likes between users and Pages $\mathbf{I}$ and the edge creation times $\mathbf{L}$

Find: Suspicious lockstep behavior - Bipartite cores of at least size (n,m) such that for each of the m Pages, all n users Liked that Page in a $2\Delta t$ time window. We call this an $[n,m,\Delta t]$-*temporally coherent bipartite core* (TBC).

We define suspicious lockstep behavior precisely below.

DEFINITION 1. *We define an $[n,m,\Delta t]$-temporally coherent bipartite core (TBC) as a set of users $\mathcal{U}' \subseteq \mathcal{U}$ and a set of Pages $\mathcal{P}' \subseteq \mathcal{P}$ such that*

$$|\mathcal{U}'| \geq n \qquad \textit{Size} \quad (1)$$

$$|\mathcal{P}'| \geq m \quad (2)$$

$$(i,j) \in \mathcal{E} \ \forall i \in \mathcal{U}', j \in \mathcal{P}' \qquad \textit{Complete} \quad (3)$$

$$\exists t_j \in \mathbb{R} \text{ s.t. } |t_j - \mathbf{L}_{i,j}| \leq \Delta t \ \forall i \in \mathcal{U}', j \in \mathcal{P}' \qquad \textit{Temporal} \quad (4)$$

We consider users in an $[n,m,\Delta t]$-TBC to be in lockstep behavior and thus suspicious.

This can be interpreted a number of different ways, each of which we will use later in the paper. From the graphical perspective, our indicator matrix $\mathbf{I}$ is an adjacency matrix for the bipartite graph between users and Pages, and our data matrix $\mathbf{L}$ contains the edge creation time. A depiction of $\mathbf{L}$ pointing out clusters of users in near temporally coherent bipartite cores can be seen in Figure 1(a-b). In terms of the graph, we have defined suspicious behavior to be bipartite cores of size greater than (n,m) in the Facebook graph where all edges going into the same Page were created in

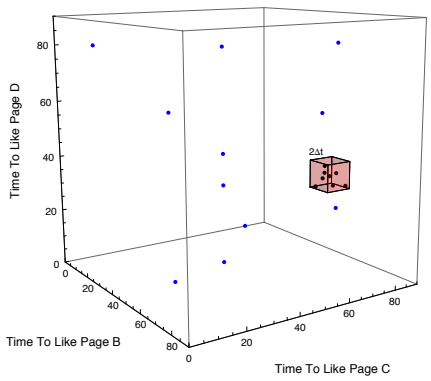


Figure 2: An example of a subspace in which we find a clear clustering of Page Likes in time. Blue dots are normal users and black dots are suspicious users part of a $[8, 3, \Delta t]$-TBC.

a small time window. This graphical view of the data and anomalous behavior is shown on a subset of the nodes in Figure 1(c).

A second interpretation can be of $\mathbf{L}$ as a data matrix where each user represents a point in M dimensional space, $\mathbf{L}_{i,*} \in \mathbb{R}^M$. (Because users do not necessarily Like all Pages, they would often fall in a subspace of the M dimensional space, but thinking about each user as a point in the M dimensional space can provide good intuition.) We then consider a group of users to be part of an $[n, m, \Delta t]$-TBC and suspicious if there exists a hypercube of width $2\Delta t$ in at least m dimensions such that at least n users fall within that hypercube. Framing the problem this way is more similar to the standard clustering literature in machine learning and the subspace clustering problem. A depiction of a 3-dimensional subspace of the M dimensional space is shown in Figure 2, where a cluster of users are all found in the same small hypercube.

Later, we will show experimentally that, for appropriate values of n, m, and Δt, such behavior is extremely uncommon and thus in fact suspicious. We additionally will show the advantage of defining it this particular way as compared to other formulations, such as only looking for bipartite cores where all edges come from one time window. This particular formulation, with constraints on the inbound edges of each node, is novel in the literature, includes these other formulations as special cases, and provides a number of advantages for preventing fraudulent behavior.

4. METHODOLOGY

With Definition 1 of suspicious lockstep behavior, the challenge remains to detect when it occurs. As shown in [22], finding bipartite cores in a graph is NP-hard. To create an algorithm to find clusters of this type, we define the problem as an optimization problem. From here we offer an iterative algorithm, which monotonically improves our results, and in Section 5 we offer an approximate MapReduce implementation that searches for many bipartite cores in parallel.

4.1 Optimization Formulation

To formulate the problem succinctly we must add additional notation. We define $\mathbf{c} \in \mathbb{R}^M$ to be a vector for the center of our cluster, such that $\mathbf{c}_j = t_j$ where t_j comes from Definition 1.

We also relax our definition of suspicious lockstep behavior to include users that are part of *temporally-coherent near bipartite cores* (TNBC). We introduce the term $\rho \in [0, 1]$, which broadly describes how many of the Page Likes a user must match (in time) to be considered suspicious. More precisely, we say that a user is suspicious if he Likes at least $\rho|\mathcal{P}'|$ of the Pages in $\mathcal{P}'$ in the designated time window. This is clearly a relaxation since all of the users in any $[n, m, \Delta t]$-TBC would also be in a $[n, m, \Delta t, \rho]$-TNBC. We give the formal definition of $[n, m, \Delta t, \rho]$-TNBC below.

DEFINITION 2. *A set of users $\mathcal{U}' \subseteq \mathcal{U}$ and a set of Pages $\mathcal{P}' \subseteq \mathcal{P}$ comprise an $[n, m, \Delta t, \rho]$-temporally coherent near bipartite core (TNBC) if there exists $\mathcal{P}'_i \subseteq \mathcal{P}'$ for all $i \in \mathcal{U}'$ such that:*

$$|\mathcal{U}'| \geq n \qquad \textit{Size} \quad (5)$$

$$|\mathcal{P}'| \geq m \quad (6)$$

$$|\mathcal{P}'_i| \geq \rho|\mathcal{P}'| \ \forall i \in \mathcal{U}' \qquad \textit{Near} \quad (7)$$

$$(i, j) \in \mathcal{E} \ \forall i \in \mathcal{U}', j \in \mathcal{P}'_i \qquad \textit{Complete} \quad (8)$$

$$\exists t_j \in \mathbb{R} \text{ s.t. } |t_j - \mathbf{L}_{i,j}| \leq \Delta t \ \forall i \in \mathcal{U}', j \in \mathcal{P}'_i \qquad \textit{Temporal} \quad (9)$$

Given these definitions our goal broadly is to maximize the number of suspicious users and the number of Page Likes of suspicious users that are suspicious (fall within the designated time window). Since we are ultimately trying to catch as many suspicious users as possible, we set $|\mathcal{P}'| = m$ and only try to grow $\mathcal{U}'$.

The optimization problem is given specifically below:

$$\max_{\mathbf{c}, \mathcal{P}' : |\mathcal{P}'| = m} \sum_i q(\mathbf{L}_{i,*} | \mathbf{c}, \mathcal{P}') \quad (10)$$

where

$$q(\mathbf{u}|\mathbf{c}, \mathcal{P}') = \begin{cases} \sigma & \text{if } \sigma = \sum_{j \in \mathcal{P}'} \mathbf{I}_{i,j} \phi(\mathbf{c}_j, \mathbf{u}_j) \geq \rho\, m \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

$$\phi(t_c, t_u) = \begin{cases} 1 & \text{if } |t_c - t_u| \leq \Delta t \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

This is a simple formulation of the problem described previously. The difference is under this formulation we are trying to find $\mathbf{c}$ and $\mathcal{P}'$ to maximize the number of users and their Likes inside the cluster centered at $\mathbf{c}$ in subspace $\mathcal{P}'$.

Additionally, because we expect most users to not be engaging in fraudulent Liking, we frame the problem similar to one-class clustering literature or to a flat kernel, where our optimization only focuses on data points *in* the cluster, and there is no penalty for data points outside the cluster.

4.2 A Serial Algorithm

To optimize this objective function, we must set both $\mathbf{c}$ and $\mathcal{P}'$. Placing $\mathbf{c}$ is similar to many density-seeking clustering problems in machine learning and data mining. Likewise, selecting $\mathcal{P}'$ from $\mathcal{P}$ is similar to subspace clustering. Therefore, we offer an iterative algorithm that in each step alternates between updating the center $\mathbf{c}$ and the subspace choice $\mathcal{P}'$, while holding the other variable constant. Note, given $\mathbf{c}$ and $\mathcal{P}'$, the users that fall within the cluster are fully determined. The algorithm can be seen in Algorithm 1.

In the step UPDATECENTER we keep $\mathcal{P}'$ constant and update $\mathbf{c}$. This update is performed iteratively over each dimension $j \in \mathcal{P}'$. For each such dimension, we find all users

$\mathcal{U}'$ that fall within the cluster but loosen the width to $\beta\Delta t$ for the current dimension we are adjusting, where $\beta > 1$ thus including users who are just outside of the time window in dimension j. Given these users we find a new center in dimension j with subroutine FindCenter. We can sort the points in $\mathcal{U}'$ based on their position in dimension j, and then in one pass, weighting users by the number of Likes from $\mathcal{P}'$ they have, find the $2\Delta t$ span for which we capture the most users and the most Likes. We use this span to update $\mathbf{c}_j$. We note that UpdateCenter runs in $O(m(mN + \log(n)))$ where we assume clusters are on the order of $O(n)$ in size.

In the step UpdateSubspace we keep $\mathbf{c}$ constant and update $\mathcal{P}'$. Given the previous values of $\mathbf{c}$ and $\mathcal{P}'$ we can find the users currently in the cluster and attempt to improve our choice of $\mathcal{P}'$, such that more Likes are included for the same current set of users in the clusters. Here we take an incremental approach. For each $j \in \mathcal{P}'$ we search among all $j' \in \mathcal{P}$. We say a user i is *covered* by a column j if $\mathbf{I}_{i,j}\phi(\mathbf{c}_j, \mathbf{L}_{i,j}) = 1$. *We only consider those columns j' for which every user covered by column j is also covered by column j'.* We can then replace column j by column j' that has the most users covered. As such, any user that was covered previously will still be covered, but we can also be adding additional coverage (for more Likes) to other users. This is not necessarily the optimal choice, but it does improve our objective and runs in $O(nmM)$ time.

We repeatedly update $\mathbf{c}$ and $\mathcal{P}'$ until neither change and the algorithm has converged, or simply for some fixed number of rounds.

4.3 Proof of Convergence

We now prove that our algorithm converges. It should be clear that our objective function is bounded, as there are a limited number of users and Page Likes, and therefore there is a maximum or set of local maxima. Therefore, we must merely show that both UpdateCenter and UpdateSubspace monotonically improve our objective function.

Lemma 1. UpdateCenter, *as defined in Algorithm 1, monotonically improves our objective function in* (10).

Proof. UpdateCenter works by updating each dimension's center one at a time, holding the others constant. For each update in each dimension, we take all the points within $\beta\Delta t$ of the previous center and find the center that will most improve our objective. Of course, all points previously covered will be included in this width of $\beta\Delta t$ since $\beta > 1$. Since we find the location for which we cover the *most* points weighted by the number of Likes for each point, we will only move the center if we find a location that covers more points with more Likes than before. Therefore, if our center moves the objective function must increase, and if it does not move then the objective function stays constant. □

Lemma 2. UpdateSubspace, *as defined in Algorithm 1, monotonically improves our objective function in* (10).

Proof. As was described previously, UpdateSubspace only replaces a $j \in \mathcal{P}'$ with a j' if all Likes covered by Page j are also covered by Page j'. Therefore, we can only improve our objective function or stay constant. Therefore UpdateSubspace monotonically improves our objective. □

Because UpdateCenter and UpdateSubspace monotonically improve our objective, the algorithm converges.

Algorithm 1 Serial CopyCatch

function S-CopyCatch($\mathbf{x}, j$)
  Require: Preset parameters Δt, n, m, and ρ
  Initialize $\mathbf{c} = \mathbf{x}$, $\mathcal{P}' = \{j\}$
  repeat
    $\mathcal{P}'_\ell = \mathcal{P}'$
    $\mathbf{c}_\ell = \mathbf{c}$
    $\mathbf{c} =$ UpdateCenter($\mathbf{c}, \mathcal{P}'$)
    $\mathcal{P}' =$ UpdateSubspace($\mathbf{c}, \mathcal{P}'$)
  until $\mathbf{c} = \mathbf{c}_\ell$ and $\mathcal{P}' = \mathcal{P}'_\ell$ // Run to convergence
return $[\mathbf{c}, \mathcal{P}]$
end function

function UpdateCenter($\mathbf{c}, \mathcal{P}'$)
  $\mathcal{U}' =$ FindUsers($\mathcal{U}, \mathbf{c}, \mathcal{P}'$) // Get current users
  Set $\mathbf{c}'$ to the average of $\mathbf{L}_{i,*}$ for all $i \in \mathcal{U}'$
  for $j \in \mathcal{P}'$ **do** // Update center for each Page
    $[\mathcal{U}', \mathbf{w}] =$ FindUsers($\mathcal{U}, \mathbf{c}, \mathcal{P}', j, \beta\Delta t$)
    $[\mathcal{U}'', t_j] =$ FindCenter($\mathcal{U}', \mathbf{w}, j$)
    $\mathbf{c}'_j = t_j$
  end for
return $\mathbf{c}'$
end function

function UpdateSubspace($\mathbf{c}, \mathcal{P}'_\ell$)
  $\mathcal{P}' = \mathcal{P}'_\ell$
  $\mathcal{U}' =$ FindUsers($\mathcal{U}, \mathbf{c}, \mathcal{P}'_\ell$) // Get current users
  for $j' \in \mathcal{P}'_\ell$ **do**
    $j'' = j'$
    $\mathcal{U}'_{j''} =$ FindUsers($\mathcal{U}', \mathbf{c}_{j''}, \{j''\}$)
    for $j \in \mathcal{P} \setminus \mathcal{P}'$ **do** // See if another Page is better
      $\mathcal{U}'_j =$ FindUsers($\mathcal{U}', \mathbf{c}_j, \{j\}$)
      if $\mathcal{U}'_{j''} \subset \mathcal{U}'_j$ **then**
        $j'' = j$, $\mathcal{U}'_{j''} = \mathcal{U}'_j$
      end if
    end for
    $\mathcal{P}' = (\mathcal{P}' \setminus \{j'\}) \cup \{j''\}$
  end for
return $\mathcal{P}'$
end function

// Find weighted center of $\mathcal{U}$ in dimension j_c
function FindCenter($\mathcal{U}, \mathbf{w}, j_c$)
  Sort $\mathcal{U}$ by $\mathbf{L}_{i,j_c}$ for $i \in \mathcal{U}$
  Scan sorted $\mathcal{U}$ for $2\Delta t$-width subset $\mathcal{U}'$
    s.t. $\sum_{i \in \mathcal{U}'} \mathbf{w}_i$ is maximized
  Set c_j to the center of this subset $\mathcal{U}'$
return $[\mathcal{U}', c_j]$
end function

// Find users from $\mathcal{U}$ based on $\mathbf{c}$ and $\mathcal{P}'$
function FindUsers($\mathcal{U}, \mathbf{c}, \mathcal{P}', j_c, \Delta t'$)
  $\mathcal{U}' = \{\}$, $\mathbf{w} = \mathbf{0}$
  for $i \in \mathcal{U}$ **do**
    for $j \in \mathcal{P}'$ **do**
      if $\mathbf{I}_{i,j} = 1 \wedge (|\mathbf{c}_j, \mathbf{L}_{i,j}| < \Delta t \vee$
        $(j = j_c \wedge |\mathbf{c}_j, \mathbf{L}_{i,j}| < \Delta t'))$ **then**
        $\mathbf{w}_i = \mathbf{w}_i + 1$
      end if
    end for
    if $\mathbf{w}_i \geq \rho m$ **then**
      $\mathcal{U}' = \mathcal{U}' \cup \{i\}$
    end if
  end for
return $[\mathcal{U}', \mathbf{w}]$
end function

5. A MAPREDUCE IMPLEMENTATION

Although Algorithm 1 works well theoretically, it has inefficiencies in both speed and convergence. To address these issues we offer here a new algorithm similar to the serial algorithm, which operates in the MapReduce framework. This implementation operates under the trade-off of making the algorithm scalable to massive data sets and trivially parallelizable, such that we can search for many clusters simultaneously, with the cost of the algorithm not provably converging. However, the heuristics used here have performed well in practice on real world data and converge quickly, as will be demonstrated in Section 7.

Unlike many optimization problems, where the goal is to find the global maximum, here we want to find all local maxima that meet our criteria (as there could be many attacks happening simultaneously with different users on different Pages). Therefore, the ability to run the algorithm from many starting points in parallel is both useful and more efficient than running from different starting points serially.

5.1 Algorithm

Because we now would like to run the algorithm for multiple clusters in parallel, we must introduce some additional notation. We define s to be the number of clusters being run simultaneously. Each cluster has a center $\mathbf{c}^{(k)} \in \mathbb{R}^M$ and a set of currently selected columns $\mathcal{P}'_k \subseteq \mathcal{P}$ (each defined as before). We define $\mathcal{C}$ to be the set of all $\mathbf{c}^{(k)}$ and $\mathcal{P}$ to be the set of all $\mathcal{P}'_k$, both for $k = 1 \dots s$.

Like the serial algorithm, the MapReduce CopyCatch algorithm operates by updating $\mathbf{c}$ and $\mathcal{P}'$ iteratively. The core of the algorithm can be seen in Algorithm 2, where we note that we run one MapReduce job per iteration, each time updating $\mathcal{C}$ and $\mathcal{P}$. As in the serial algorithm, we can keep iteratively updating $\mathcal{C}$ and $\mathcal{P}$ until no changes are made. In practice, we will merely run a fixed number of iterations.

MapReduce: It is worth taking a moment to note the data flow in a MapReduce job before describing the details of our algorithm. In the Map step, our input is split among many mappers. Each mapper gets a pair of data of the form $\langle \text{KEY}_{\text{map}}, \text{VALUE} \rangle$ and can output zero or more results of the form $\langle \text{KEY}_{\text{reduce}}, \text{VALUE} \rangle$. In the reducer step, for each unique $\text{KEY}_{\text{reduce}}$ a reducer is formed which takes as an input $\langle \text{KEY}_{\text{reduce}}, \text{VALUES} \rangle$, where VALUES is a set of the VALUE outputs from the mapper step which correspond to that reducer's particular $\text{KEY}_{\text{reduce}}$. The reducer can then output data to disk. Aside from this data flow, we make use of Hadoop's Distributed Cache, which lets us store data as global read-only data. For more information on MapReduce and Hadoop see [7, 1].

In our implementation, the mapper for the MapReduce job UserMapper is shown in Procedure 3, the reducer AdjustCluster-Reducer is shown in Procedure 4, and $\mathcal{C}_\ell$ and $\mathcal{P}_\ell$ are stored in the Distributed Cache.

UserMapper finds which users are currently in which clusters based on $\mathcal{C}$ and $\mathcal{P}$ and maps those users to a reducer based on which cluster it is within. More specifically, the Map step takes as input $\mathbf{L}$ and $\mathbf{I}$, where each $(\mathbf{L}_{i,*}, \mathbf{I}_{i,*})$ for all $i \in \mathcal{U}$ is input to a mapper. Each mapper checks the $\mathbf{L}_{i,*}$ across all s clusters to see if it falls within that cluster following the definition given in our optimization objective (10). If it does, it emits an output where the key is the cluster ID k, and the value is the row (user) information $\mathbf{L}_{i,*}$ and $\mathbf{I}_{i,*}$. Each mapper runs in $O(sm)$ time, and since this is being run over all data the entire step takes $O(smN)$ not taking into account parallelization.

AdjustCluster-Reducer takes in all of the users currently in a given cluster k and updates $\mathbf{c}^{(k)}$ and $\mathcal{P}'_k$. Each reducer takes as an input $\langle k, \mathcal{U}' \rangle$, where $\mathcal{U}'$ here contains pairs $(\mathbf{L}_{i,*}, \mathbf{I}_{i,*})$ for all users in the cluster (as was output by the mappers). As shown in Procedure 4, we must be careful to only use values from each user in the dimensions for which it falls within the cluster. However, beyond this, the update generally works fairly simply. The center $\mathbf{c}$ is updated by merely taking an average of the points in the cluster, similar to a mean-shift algorithm with a flat kernel [5]. The selected columns are chosen based on which columns cover the most users from the previous cluster parameters, and then by which columns have the lowest variance among these users. By using the previous centers for this update, we can do the calculation online, passing over each user only once. Because we assume each cluster is $O(n)$ in size, each reducer takes $O(nM)$ time, and the reduce step as a whole takes $O(snM)$ when not considering parallelization. The reducers output the updated $\mathcal{C}$ and $\mathcal{P}$ to be placed in the Distributed Cache in subsequent iterations.

Algorithm 2 MapReduce CopyCatch

Require: Preset parameters Δt, m, and ρ
$\mathcal{C}, \mathcal{P} =$ Initialize()
repeat
  $\mathcal{C}_\ell = \mathcal{C}$, $\mathcal{P}_\ell = \mathcal{P}$
  $\mathcal{C}, \mathcal{P} =$ MapReduceJob$(\mathcal{C}_\ell, \mathcal{P}_\ell)$
until $\mathcal{C}_\ell = \mathcal{C} \wedge \mathcal{P}_\ell = \mathcal{P}$
return $[\mathcal{C}, \mathcal{P}]$

Procedure 3 UserMapper$(\langle \text{Null}, (\mathbf{L}_{i,*}, \mathbf{I}_{i,*}) \rangle)$

Globals: $\mathcal{C}, \mathcal{P}$
for $k = 1 \dots s$ **do**
  $\sigma = \sum_{j \in \mathcal{P}'_k} \mathbf{I}_{i,j} \cdot \phi(\mathbf{c}_j^{(k)}, \mathbf{L}_{i,j})$
  if $\sigma \geq \rho |\mathcal{P}'_k|$ **then**
    emit $\langle k, (\mathbf{L}_{i,*}, \mathbf{I}_{i,*}) \rangle$
  end if
end for

Procedure 4 AdjustCluster-Reducer$(k, \mathcal{U}')$

Globals: $\mathcal{C}, \mathcal{P}$
Initialize $\mathbf{c} = \mathbf{0}$, $\mathbf{p} = \mathbf{0}$, $\mathbf{v} = \mathbf{0}$
for all map values $(\mathbf{L}_{i,*}, \mathbf{I}_{i,*}) \in \mathcal{U}'$ **do**
  for $j = 1 \dots M$ **do**
    if $\mathbf{I}_{i,j} = 1 \wedge \phi(\mathbf{c}_j^{(k)}, \mathbf{L}_{i,j}) = 1$ **then**
      $\mathbf{c}_j = \mathbf{c}_j + \mathbf{L}_{i,j}$
      $\mathbf{p}_j = \mathbf{p}_j + 1$
      $\mathbf{v}_j = \mathbf{v}_j + (\mathbf{c}_j^{(k)} - \mathbf{L}_{i,j})^2$
    end if
  end for
end for
$\mathbf{c}^{(k)} = \mathbf{c}/\mathbf{p}$
$\mathbf{v} = \mathbf{v}/\mathbf{p}$
Sort $\{j\}_1^M$ by $\mathbf{p}$ (decreasing), then $\mathbf{v}$ (increasing)
Set $\mathcal{P}'_k$ to top m columns from previous sort
return Updated $\mathbf{c}^{(k)}$ and $\mathcal{P}'_k$

5.2 Implementation Optimizations

While the description above gives the general overview of the algorithm, there are a number of implementation details that make the algorithm run efficiently on huge data sets.

Data Format Because $\mathbf{L}$ is expected to be very large and sparse, we do not want to, nor need to, store the full data matrix. Instead, we store the matrix as an adjacency list. For each user $i \in \mathcal{U}$, we store on one line the user ID i, a list of page IDs j where $\mathbf{I}_{i,j} = 1$, and the value $\mathbf{L}_{i,j}$. As a result, NULL values were $\mathbf{I}_{i,j} = 0$ do not take up space or time.

$\mathcal{C}$ and $\mathcal{P}$ are stored similarly where each line of a data file is indexed by the cluster ID k, and then contains the Page IDs j, the times $\mathbf{c}_j^{(k)}$ and a 1 if $j \in \mathcal{P}'_k$.

Seeds and Initial Iterations Figuring out where to start the clusters can be very difficult. To avoid any bias, we sample seeds randomly from the list of all edges in the graph, using the edge's Page and Like time to initialize both $\mathcal{P}'$ and $\mathbf{c}$. (While we could use suspicious users from one of Facebook's many other security mechanisms, this would introduce prior assumptions about attackers that are unnecessary and could make it easier for an adversary to hide.)

As a result, in the initial iterations users only need to have Liked a single Page at around the same time, which is not uncommon. There are often so many users found in these initial iterations that we sample a small percentage of them at random to keep the algorithm efficient. Even with the sampling, this method lets us quickly find lots of users that Liked one Page around the same time, and then see what else they have in common. This sampling is performed in the first two iterations until $|\mathcal{P}'| = m$.

Page sampling Because $\mathcal{C}$ and $\mathcal{P}$ are stored in the Distributed Cache, they are passed to every MapReduce node. This communication time slows the algorithm down if the data becomes too large. To avoid this we limit the length of $\mathbf{c}^{(k)}$ by including only Pages from $\mathcal{P}'_k$, Pages that were close to being in $\mathcal{P}'_k$ in the last iteration, and a random sampling of the other Pages. With this method we can still find users that are similar, but without the risk of becoming too slow.

6. AN ADVERSARIAL CHALLENGE

Given our definition of suspicious lockstep behavior and our approach to detecting spam, how much damage could an adversary do without appearing suspicious? We will again discuss this in the context of Facebook, although it can be extended to other applications.

To be more concrete, we can frame the question as follows: If an adversary controls N' accounts and wants to Like M' Pages, for $N \gg N' \gg n$ and $M \gg M' \gg m$, how long would it take the adversary to Like all M' Pages with all N' accounts without creating an $[n, m, \Delta t, \rho]$-TNBC? (For this analysis we assume that we catch all lockstep behavior meeting Definition 2.) It turns out this is an extension of an old open problem in extremal graph theory. We analyze below a couple different approaches to this problem.

6.1 "Greedy Attacks"

We first analyze the consequences of an adversary performing a naïve greedy attack, particularly because it matches a common business model for adversaries, demonstrates the difficulties for an adversary, and shows the strength of our security approach.

In this initial example, we assume that the adversary has from the start the set of M' Pages he wants to Like, and we analyze the effect if he iteratively Likes each Page with as many accounts as he can without getting caught. That is, for the first Page the adversary will Like the Page using as many accounts as possible without getting caught. He will then move onto the second Page and do the same thing, etc. In this case, we are unconcerned with the time component of our bipartite core definition because we assume the adversary will add all of the Likes instantaneously. Therefore, an adversary will be caught if he creates an (n, m) complete bipartite core in the graph. If we look at the $N' \times M'$ adjacency matrix of Likes added by the adversary, this is equivalent to creating an n, m submatrix filled with ones. We will call this $N' \times M'$ adjacency matrix $\mathbf{I}'$.

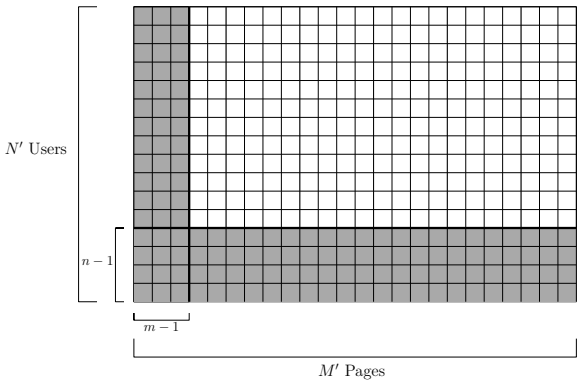


Figure 3: Illustration of a greedy attack - $N' \times M'$ adjacency matrix $\mathbf{I}'$ of all Likes, where grey cells denote $\mathbf{I}'_{i,j} = 1$, and white cells denote $\mathbf{I}'_{i,j} = 0$.

LEMMA 3. *We assume there is an adversary with N' accounts performing a greedy attack on M' Pages. Catching and preventing all behavior meeting Definition 1 limits the adversary to obtaining $(m-1)N' + (n-1)M' - (n-1)(m-1)$ Likes at a rate greater than $\frac{n-1}{2\Delta t}$ per Page per unit time.*

PROOF. A simple pattern created by following the greedy approach can be seen in Figure 3. We see here for the first $m-1$ pages, the adversary can Like the Page with all N' accounts. This is because we will only detect the accounts if at least m pages have been Liked, so there is no harm in Liking the first $m-1$ Pages with as many accounts as possible. However, for the mth Page, if any n users Like that page in a $2\Delta t$ time window, then he will have created an $n \times m$ full submatrix. This is true for all Pages $\geq m$. As a result, following this greedy approach we can only Like each Page an average of $\frac{(m-1)N' + (n-1)M' - (n-1)(m-1)}{M'}$ times. □

While not a complex example, the results are quite interesting. First, we see that as an adversary adds more accounts, the average number of Likes added by each account asymptotically approaches $m-1$. Similarly, as the accounts Like more Pages, the average number of Likes per Page asymptotically approaches $n-1$.

We have so far focused on how many Likes can an adversary add instantaneously, but it is worth looking at the effect of our time constraint. To illustrate the impact of our temporal coherence restriction, we look at the case where an adversary sells Likes on eBay, promising to add the Page Likes within some time period (as some adversaries are known to do). As new orders come in, the adversary must add those Page Likes within the requested amount of time. Because

we have a distinct time window for each Page j (centered around $\mathbf{c}_j$), waiting to add Likes to a new Page does not help an adversary avoid being caught. That is, if an adversary gets $m-1$ orders in January and adds all N' Likes as the greedy attack suggests, then even if he does not get any more orders until July he still cannot add more than $n-1$ Likes in a $2\Delta t$ time window without being caught.

This is in contrast to previous research, which could set a time window for the entire bipartite core (through limiting the input), and thus merely waiting out that time window gave accounts a clean slate. As a result of our construction, once the greedy attack has been run, the adversary can only continue to add Page Likes to each Page at a rate slower than $\frac{n}{2\Delta t}$. We call this effect *multi-user rate limiting*.

To make the effect of these restrictions clear and concrete, let's look at the example limits $n = 20$ accounts, $m = 5$ Pages, and $2\Delta t =$ one week. (Note, for security reasons these are *not* the actual limits used at Facebook.) Any adversary performing a greedy attack with $N' = 1000$ accounts can on average only like up to 5 Pages per account for 980 of his accounts. Similarly, if the adversary wants to boost the Like counts for more than 5 Pages, he can on average only Like the Pages 20 times. To add additional likes to any given Page can only be done at a rate slower than $\frac{20 \text{ accounts}}{1 \text{ week}} \approx 3$ Likes per Page per day. This is clearly much harsher rate-limiting than we could enforce on a single user, but when looking at the group acting together makes sense.

6.2 Optimal Strategy: An Open Problem

It should be clear from the previous analysis that greedy attacks do not work well for spammers. Ideally, we would like to find an upper bound for the amount of damage an adversary could inflict. However, this turns out to be an old, open problem in extremal graph theory. In 1951 Kazimierz Zarankiewicz posed the following problem [27]: how many edges can there be in a bipartite graph G of size $N' \times M'$ without creating a complete bipartite core of size $n \times m$? This is known as the Zarankiewicz problem, and the maximum number of edges is denoted as $z(N', M', n, m)$.

Although the problem is old, little progress has been made in solving it for the general case. Füredi [13] is currently known to have the best general upper bound

$$z(N', M', n, m) \leq (n-m+1)^{\frac{1}{m}} N' M'^{1-\frac{1}{m}} + M'm + mN'^{2-\frac{2}{m}}$$

for $n > m$. However, this is only known to be asymptotically optimal for $m = 2$ and $n = m = 3$. If we use larger values of n and m and reasonable values of N' and M', then $z > N'M'$ and thus offers us no information.

Additionally, the proof offered by Füredi [13], as well as most work surrounding the Zarankiewicz problem, uses non-constructive methods. That is, although the proof finds an upper bound for $z(N', M', n, m)$, it does not give any information about how to actually add edges to reach that bound without being caught. It would therefore be a challenge for an adversary to optimally add edges (and would be interesting for the field of extremal graph theory if solved).

It is worth noting that our problem deviates from the classic Zarankiewicz problem in two regards: (1) we look for near-bipartite cores, and (2) we require temporal coherence. With respect to the first point, by looking for near bipartite cores and not just complete bipartite cores, we would also catch cases where certain 1's were missing from the $n \times m$ submatrix of $\mathbf{I}'$. We call this the *approximate Zarankiewicz problem*. While any upper bound for the Zarankiewicz problem is also an upper bound to this approximate Zarankiewicz problem, the maximum number of edges added without creating a $[n, m, \Delta t, \rho]$-TNBC would be lower, and thus this is an even harder problem for an adversary.

Second, we require temporal coherence in our bipartite cores for the behavior to be considered suspicious. This helps us more accurately and flexibly discern normal behavior from illegitimate behavior and effectively rate limits an adversary, forcing them to add Likes very slowly if they do not want to be caught. As we saw in our analysis of a greedy attack, this multi-user rate limiting is significantly stricter than a rate limit we could enforce on individual users. When generalizing this concept to the Zarankiewicz problem, the constraint adds complexity to an already challenging problem. While we cannot find a precise optimal rate at which adversaries can add edges without knowing an optimal strategy for the Zarankiewicz problem, it should be clear that the multi-user rate limiting principle holds. Setting $2\Delta t = \infty$ restricts an adversary to solving the Zarankiewicz problem; decreasing Δt allows an adversary to exceed the maximum solution to the Zarankiewicz problem but at a slow rate.

7. EXPERIMENTAL ANALYSIS

7.1 Experimental Setup

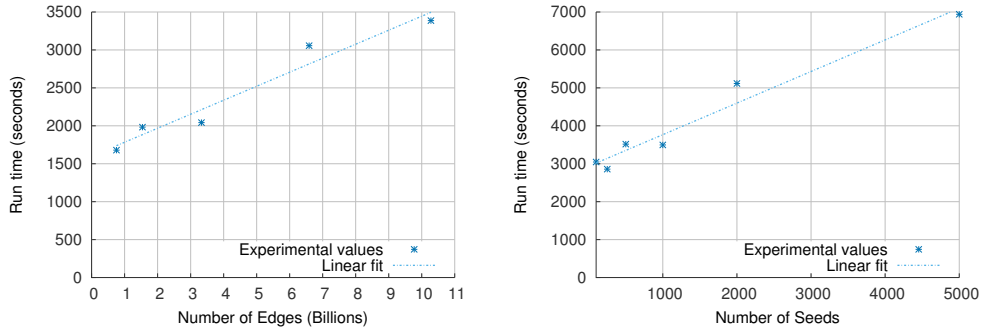
CopyCatch was written in Java 1.6.0_14 with Hadoop 0.20.1, generally matching the algorithm outlined in Section 5. The experiments were run on one of Facebook's Hadoop clusters, running Hadoop, Hive, and HDFS on over 1000 machines. More information about Facebook's infrastructure can be found in [26]. Our MapReduce jobs ran with 3000 mappers and 500 reducers. We ran the algorithm on a few different datasets from Faceboook as well as synthetic data to demonstrate a number of different properties.

The Facebook datasets used come from real Likes between users and Pages on the site. We pull data from periods of time ranging from weeks to multiple months, where the data has not already had Likes removed by this particular method (although of course many other security measures already keep malicious users and fake Likes off the site). For our scalability tests, data ranged from approximately 760 million Page Likes (25 gigabytes on HDFS) to 10.4 billion Page Likes (294 gigabytes on HDFS). When not testing scalability over data size, we used an intermediate dataset of 3.3 billion Likes (100 gigabytes on HDFS). Our parameter choices for n, m, and Δt are those used currently on Facebook systems, but can not be given for security reasons.

We also ran our discoverability experiments on synthetic data so that results could be replicated by other researchers. Our synthetic data was generated following the RTM method [2], where the time evolving graph is generated with a repeated Kronecker product. The code used to generate the time evolving graph, including our initial generator/tensor, can be found online at `alexbeutel.com/l/www2013`. The generated graph is a bipartite graph between 38 million and 10 million nodes with 410 million edges. After our data formatting, the graph is 10 gigabytes on HDFS.

7.2 Scalability

Facebook now has over a billion users and is continuing to grow. Therefore, it is important that our algorithm scales well to large datasets. We test this a few different ways.



(a) (b)

Figure 4: Scalability experiments: (a) shows the linear increase in computation time as the graph grows into the billions of Page Likes, (b) shows the increase in computation time as the number of seeds increases. Running time for is based on the total time for the first three iterations of CopyCatch.

A necessary preprocessing step to keep our algorithm efficient is to format the data similar to an adjacency list as described in Section 5.2. This takes approximately 45 minutes for our 100GB Facebook dataset. Because this is merely formatting and not required each time we run the algorithm, we do not consider this time in future tests.

For each of our timing experiments, we run our algorithm on the on the 100GB Facebook dataset and time the first 3 iterations, which includes the initial iterations of starting with individual Likes as seeds. We chose to run these tests on the Facebook data because run time is heavily influenced by the size of the data and the size of clusters. Since we do not know a-priori where large clusters are, we sample our seeds randomly so that we get the same distribution of small and large clusters that we would get in our real runs.

Even with large Hadoop clusters, it is important that the algorithm can scale as the data scales. We test this by running our implementation over Page Likes from increasingly large periods of time. As we explained previously, this data ranges from 25GB with 760 million Page Likes to 294GB with 10.4 billion Page Likes. We ran the algorithm with 100 seeds, 3000 mappers, and 500 reducers. The run time for the first 3 iterations can be seen in Figure 4(a). As seen in the plot, the run time increases approximately linearly with the number of edges (Page Likes). However, we note that only after a greater than 10 times increases in data size does our running time double. Therefore as our data grows, the running time of CopyCatch grows at a much slower rate.

As the data scales, we also face the challenge of having to use more seeds to sufficiently sample the space. Therefore, we also tested how the run time increases as the number of clusters being run in parallel increases. We time the first 3 iterations for running the algorithm on the 100GB Facebook dataset with 3000 mappers and 500 reducers. We vary the number of seeds used from 100 to 5000. As can be seen in Figure 4(b), we find a similar linear relationship between number of seeds and run time. We note that 50 times increase from 100 clusters to 5000 clusters takes only a little over twice as much time. This is again reassuring that our implementation exploits the parallelism of the problem and can continue to scale as the data scales.

7.3 Convergence

Because our MapReduce implementation is not provably convergent, we also test our algorithm's convergence. We ran the algorithm on the 100GB Facebook dataset, starting with 1000 seeds and tracked the sum of the values of the objective functions per iteration, starting at the third iteration when $|\mathcal{P}'| = m$. As can be seen in Figure 5, the algorithm quickly converges in only a few iterations as desired.

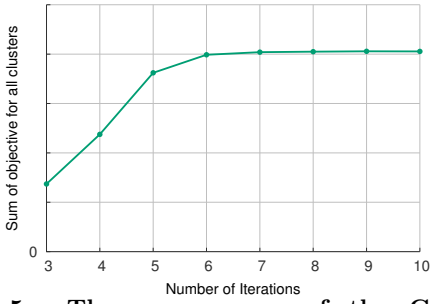


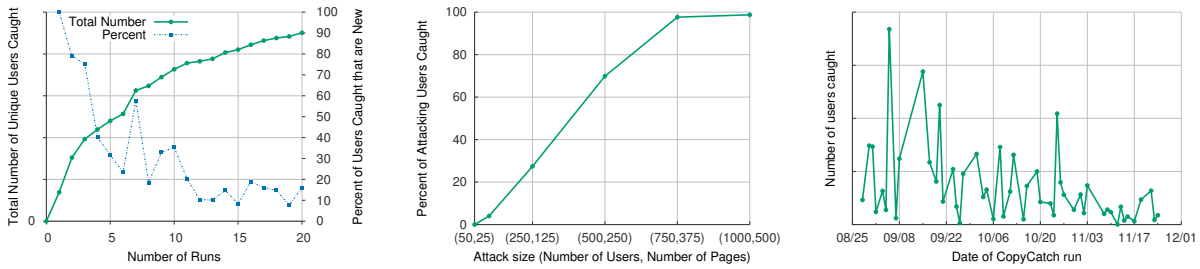
Figure 5: The convergence of the CopyCatch MapReduce implementation over 10 iterations on Facebook data.

7.4 Discovery

The effectiveness of the algorithm is measured by its ability to detect fraudulent behavior. This is difficult to gauge in the real world where there isn't labelled data, and it can be impossible to know if a user intentionally, honestly Liked a given Page. Here we take two different approaches to evaluate our method.

First, we analyze the success of the algorithm in finding suspicious groups of users on Facebook. Our goal is for the algorithm to find as many of the $[n, m, \Delta t, \rho]$-TNBC's in the Facebook data as possible. To test CopyCatch's success, we run the algorithm repeatedly and check that we eventually are mostly finding users we had caught in previous runs. In our experiment we run the algorithm 20 times for 5 iterations, each run starting from 1000 seeds on the 100GB data set. In Figure 6(a) we see that over the course of 20 iterations we quickly decrease to finding mostly the same users repeatedly, and only catching a small percentage of new users with each successive run. More precisely, after the eleventh run, the average percent of caught users that are new is only 13% percent. Because we use random independently drawn seeds for each run, this test suggests that the algorithm has found most of the attacks in the dataset.

For a more precise analysis we use our synthetic dataset to test the algorithm's ability to find attacks in the graph. We set $n = 50$, $m = 25$, $\Delta t = 50$, and $\rho = 0.9$. For our



(a) Facebook (b) Synthetic (c) Facebook live runs

Figure 6: Plots demonstrating the effectiveness of CopyCatch in discovering attacks. (a) shows the total number of users caught after multiple runs of CopyCatch. (b) shows the success in finding planted attacks in synthetic data after one run. (c) shows the decrease in attacks on Facebook over the last 3 months.

tests we add 20 randomly-placed attacks to the graph, and test the ability of the algorithm to find the attacks after one run starting with 5000 randomly chosen seeds. We vary our attack size as a multiple of our defined suspicious lockstep behavior, ranging from 50 users and 25 Pages to 1000 users and 500 Pages. In each case the time of the attack for each Page is chosen at random, and each user's Likes falls within the $2\Delta t$ time window for 95% of the Pages being attacked. The percent of attackers caught after 1 run for each attack size are plotted in Figure 6(b). Also we note that 0% of our caught users were false positives. As we see in the plot, small attacks exactly at our threshold size were hard to spot, but as attacks grow in size in just one run we catch nearly all of the attackers. It is worth noting that because we choose seeds randomly, running the algorithm again should catch an independent set of the attackers, and thus merely running the algorithm a few times should catch a high percentage of the attackers even in cases where 1 run does not. Overall, this experiment shows the algorithm is generally successful in detecting a few relatively small attacks in large graphs.

7.5 Deployment at Facebook

CopyCatch is run regularly at Facebook, searching for new attacks. Parameters have been chosen to significantly distinguish natural user behavior from ill-gotten Page Likes. In practice at Facebook, false positives are very rare due to the sparsity of the Page Like matrix. In particular, we labelled 22 randomly selected clusters caught in February 2013. After intensive manual investigation, we found that 100% of the clusters were caught due to Likes generated through deceitful means - 5 from fake accounts, 13 from malicious browser extensions, 1 from OS malware, 2 from credential stealing, and 1 from social engineering. When caught, users that have contributed to ill-gotten Page Likes have a portion of their past month's Likes removed and are prevented from Liking pages for the next month. This removal of Page Likes are reflected in the Like counts of the Pages that benefited. As shown in Figure 6(c), we have seen a general decrease in ill-gotten Page Likes since the start of this method. Overall, this method, in combination with the numerous other measures at Facebook mentioned in Section 1, has proved effective in decreasing ill-gotten Page Likes.

8. DISCUSSION: APPLICATIONS

As mentioned previously, the algorithm can be used in a number of settings including Twitter followers and Amazon product reviews. The use case at Twitter demonstrates the ability of the algorithm to be applied to non-bipartite graphs. However, in this case the general structure of the algorithm is the same - we look for some set of users that follow another set of users at around the same time.

In this case of product reviews, we again have a bipartite graph between users and products where edges represent a review that was given to the product. However, in the case of product reviews, edges have much more meta-information beyond just timestamps, such as the IP address the review was written from, review tone, and linguistic cues. While this paper describes the edge constraints in terms of time, the algorithm only requires some center $\mathbf{c}$ and a comparison function ϕ. Therefore, we could extend the formulation as applied to Page Likes to use multiple linguistic and behavioral cues when attempting to find near bipartite cores in settings with more metadata.

9. CONCLUSIONS

In this paper we have attacked the problem of detecting fraudulent user feedback through the context of catching lockstep behavior in Facebook Page Likes. Our main contributions are:

1. **Problem Formulation:** We offer a novel definition of suspicious behavior based only on graph structure and edge constraints.
2. **Algorithm:** We describe two new algorithms to find lockstep behavior ($[n, m, \Delta t, \rho]$-TNBC's) - one provably convergent serial algorithm and one scalable, efficient MapReduce implementation.
3. **Theoretical analysis:** We show that catching lockstep behavior limits the damage an adversary can do. By analyzing the effect of a "greedy attack" and by applying the 60 year old Zarankiewicz problem to our setting, we show that finding an optimal adversarial attack is very hard.

Finally, we experimentally demonstrate on Facebook and synthetic data that CopyCatch is scalable, generally converges, and effective in catching lockstep behavior.

Acknowledgments: We would like to thank Dr. Tao Stein, Kristie Chow, and Jieqi Yu for their valuable help and guidance in pursuing this project while at Facebook, and Professor Geoff Gordon and Dr. Stephan Günnemann for their interesting and insightful discussions leading up to this paper.

This material is based upon work supported by Facebook, a National Science Foundation Graduate Research Fellowship (Grant No. DGE-1252522), and the National Science Foundation under Grant No. IIS-1017415.

10. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/, 2012.

[2] L. Akoglu, M. Mcglohon, and C. Faloutsos. RTM: Laws and a recursive generator for weighted time-evolving graphs. In *International Conference on Data Mining*, December 2008.

[3] A. Anagnostopoulos, A. Dasgupta, and R. Kumar. Approximation algorithms for co-clustering. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '08, pages 201–210, New York, NY, USA, 2008. ACM.

[4] A. Banerjee, I. Dhillon, J. Ghosh, S. Merugu, and D. Modha. A generalized maximum entropy approach to bregman co-clustering and matrix approximation. *The Journal of Machine Learning Research*, 8:1919–1986, October 2007.

[5] Y. Cheng. Mean shift, mode seeking, and clustering. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8):790 –799, aug 1995.

[6] K. Crammer and G. Chechik. A needle in a haystack: local one-class optimization. In *Proceedings of the twenty-first international conference on Machine learning*, ICML '04, pages 26–, New York, NY, USA, 2004. ACM.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI'04*, Dec. 2004.

[8] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *Conference of the ACM Special Interest Group on Knowledge Discovery and Data Mining*, New York, NY, 2003. ACM Press.

[9] Facebook. Better Security through Software. blog.facebook.com/blog.php?post=248766257130. 2010.

[10] Facebook. Working Together to Keep You Secure. blog.facebook.com/blog.php?post=68886667130, 2009.

[11] Facebook. Staying in Control of Your Facebook Logins. blog.facebook.com/blog.php?post=389991097130, 2010.

[12] Facebook. Improvements to our Site Integrity Systems. facebook.com/10151005934870766, 2012.

[13] Z. Füredi. An upper bound on Zarankiewicz' Problem. *Combinatorics, Probability and Computing*, 5(01):29–33, 1996.

[14] T. George and S. Merugu. A scalable collaborative filtering framework based on co-clustering. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 625–628, Washington, DC, USA, 2005. IEEE Computer Society.

[15] G. Gupta and J. Ghosh. Robust one-class clustering using hybrid global and local search. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 273–280, New York, NY, USA, 2005. ACM.

[16] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1):1:1–1:58, Helen Martin 2009.

[17] K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *Proceedings of the Third International Conference on Advances in Social Network Analysis and Mining*, 2011.

[18] S. Pandit, D. Chau, S. Wang, and C. Faloutsos. Netprobe: a fast and scalable system for fraud detection in online auction networks. In *Proceedings of the 16th international conference on World Wide Web*, pages 201–210. ACM, 2007.

[19] S. Papadimitriou and J. Sun. Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *Data Mining, 2008. ICDM '08. Eighth IEEE International Conference on*, pages 512 –521, dec. 2008.

[20] E. Papalexakis, A. Beutel, and P. Steenkiste. Network anomaly detection using co-clustering. In *2012 International Conference on Advances in Social Network Analysis and Mining, ASONAM 2012*, 2012.

[21] E. Papalexakis and N. Sidiropoulos. Co-clustering as multilinear decomposition with sparse latent factors. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 2064–2067. IEEE, 2011.

[22] R. Peeters. The maximum edge biclique problem is np-complete. *Discrete Appl. Math.*, 131(3):651–654, Sept. 2003.

[23] J. Pei, D. Jiang, and A. Zhang. On mining cross-graph quasi-cliques. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, pages 228–238, New York, NY, USA, 2005. ACM.

[24] B. A. Prakash, M. Seshadri, A. Sridharan, S. Machiraju, and C. Faloutsos. Eigenspokes: Surprising patterns and scalable community chipping in large graphs. *PAKDD 2010*, 21-24 June 2010.

[25] T. Stein, E. Chen, and K. Mangla. Facebook immune system. In *Proceedings of the 4th Workshop on Social Network Systems*, SNS '11, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[26] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1013–1020, New York, NY, USA, 2010. ACM.

[27] K. Zarankiewicz. Problem p 101. In *Colloq. Math*, volume 2, page 301, 1951.