ORFEL: efficient detection of defamation or illegitimate promotion in online recommendation

Gabriel Gimenes,
Robson L F Cordeiro
and Jose F Rodrigues-Jr

University of Sao Paulo
Av Trab Sao-carlense, 400
Sao Carlos, SP, Brazil-13566-590
{*ggimenes, robson, junio*}*@icmc.usp.br*

Abstract

What if a successful company starts to receive a torrent of low-valued (one or two stars) recommendations in its mobile apps from multiple users within a short (say one month) period of time? Is it legitimate evidence that the apps have lost in quality, or an intentional plan (via lockstep behavior) to steal market share through defamation? In the case of a systematic attack to one's reputation, it might not be possible to manually discern between legitimate and fraudulent interaction within the huge universe of possibilities of user-product recommendation. Previous works have focused on this issue, but none of them took into account the context, modeling, and scale that we consider in this paper. Here, we propose the novel method Online-Recommendation Fraud ExcLuder (*ORFEL*) to detect defamation and/or illegitimate promotion of online products by using vertex-centric asynchronous parallel processing of bipartite (users-products) graphs. With an innovative algorithm, our results demonstrate both efficacy and efficiency – over 95% of potential attacks were detected, and *ORFEL* was at least two orders of magnitude faster than the state-of-the-art. Over a novel methodology, our main contributions are: (1) a new algorithmic solution; (2) one scalable approach; and (3) a novel context and modeling of the problem, which now addresses both defamation and illegitimate promotion. Our work deals with relevant issues of the Web 2.0, potentially augmenting the credibility of online recommendation to prevent losses to both customers and vendors.

Keywords: `graphs, fraud detection, defamation, recommendation, Web 2.0, data analysis`

1. Introduction

In the Web 2.0, it is up to the users to provide content, like photos, text, recommendations and many other types of user-generated information. The more interaction, e.g., likes, recommendation, comments, etc., a product page (or a user profile) gets, the

better are the potential profits that a company (or an individual) may achieve with automatic recommendation, advertisement, and/or priority in automatic search engines. In Google Play, for example, mobile apps heavily depend on high-valued (4 or 5 stars) recommendations to get more important and to expand their pool of customers; on Amazon, users are offered the most recommended products, that is, those that were better rated; and in TripAdvisor, users rely on other's feedback to pick their next travels. The same holds for defamation, which is the act of lowering the rank of a product by creating artificial, low-valued recommendations. Sadly, fraudulent interaction has come up in the Web 2.0 – fraudulent likes, recommendations, and evaluations define artificial interests that may illegitimately induce the importance of online competitors.

Attackers create illegitimate interaction by means of fake users, malware credential stealing, Web robots, and/or social engineering. The identification of such behavior has great importance to companies, not only because of the potential losses due to fraud, but also because their customers tend to consider the reliability of a given website as an indicator of trustfulness and quality. According to Facebook [11], fraudulent interaction is harmful to all users and to the Internet as a whole, so it is important that users have a true engagement around brands and content.

However, catching up with such attacks is a challenging task, especially when there are millions of users and millions of products being evaluated in a system that deals with billions of interactions per day. In such attacks, multiple fake users interact with multiple products at random moments [1] in a way that their behavior is camouflaged among millions of legitimate interactions per second. The core of the problem is: *how to track the temporal evolution of fraudulent user-product activity since the number of possible interactions is factorial?*

We want to identify the so-called *lockstep behavior*, i.e., groups of users acting together, generally interacting with the same products at around the same time. As an example, imagine that an attacker creates a set of fake users to artificially promote his e-commerce website; then, he would like to comment and/or recommend his own Web pages, posts, or advertisements to gain publicity that, fairly, should come from real customers. Here, an attacker may refer to employees related to a given company, professionals (spammers) hired for this specific kind of job, Web robots, or even anonymous users. The weak point in all these possibilities is that the attacker must substantially interact with the attacked system within limited time windows; also, the attacker must optimize his efforts by using each fake user account to interact with multiple products. This behavior agrees with the lockstep definition. Note that this pattern is well-defined in online recommendation and in many other domains, such as academic co-citation, social network interaction, and search-engine optimization. Provided that this is not a new problem, we use in this paper the definition of *lockstep behavior* given by Beutel *et al.* [4]. See the upcoming Section 3.3 for details.

The task of identifying locksteps is commonly modeled as a graph problem – nodes are either users or products; weighted edges represent recommendations – in which we want to detect near-bipartite cores considering a given time constraint. The bipartite cores correspond to groups of users that interacted with groups of products within limited time intervals. One lockstep may be defamation, when the interactions are negative (low-valued) recommendations; or illegitimate promotion, when the recommendations are positive. Therefore, the problem generalizes to finding near-bipartite cores with

edges whose weights correspond to the rank of the recommendations. Note that we want to tackle the problem *without* any previous knowledge about suspicious users, products, nor the moments when frauds occurred in the past.

This work extends the state-of-the-art solutions for the problem of lockstep identification. Our main contributions are threefold:

1. **Novel algorithmic paradigm**: we introduce the first *vertex-centric* algorithm able to spot *lockstep behavior* in Web-scale graphs using asynchronous parallel processing; vertex-centric processing is a promising paradigm that still lacks algorithms specifically tailored to its *modus operandi*;
2. **Scalability and accuracy**: we tackle the problem for billion-scale graphs in one *single* commodity machine, achieving efficiency that is comparable to that achieved by state-of-the-art works on large *clusters* of computers, whilst obtaining the same efficacy;
3. **Generality of scope:** we tackle the problem for real weighted graphs ranging from social networks to e-commerce recommendation, expanding the state-of-the-art of lockstep semantics to discriminate defamation *and* illegitimate promotion.

This paper follows a traditional organization. Section 2 presents background concepts, while Section 3 reviews the related works. Our proposal is described in Section 4. In Section 5, we report experimental results, including real data analyses. Finally, Section 6 concludes the paper and presents ideas for future work.

2. Background

2.1. Vertex-centric graph processing

We use in this paper the well-known concept of vertex-centric processing [23]. Given a graph $G = (V, E)$ with vertices labeled from 1 to $|V|$, we associate a value to each vertice and to each edge – for a given edge $e = (u, v)$, u is the source and v is the target. With values associated to vertices and edges, vertex-centric processing corresponds to the *graph scan* approach depicted in Algorithm 1. The values are determined according to the computation that is desired, e.g., Pagerank or belief propagation; we illustrate this fact with hypothetical functions f and g in the algorithm. Evidently, a single scan is not enough for most useful computations, therefore, the graph is commonly scanned many times until a criterion of convergence is satisfied. Graph processing, then, becomes what is defined in Algorithm 2.

The vertex-centric processing paradigm contrasts with usual graph traversal algorithms, like breadth-first or depth-first searches. While traversal-based algorithms support any kind of graph processing, they are made to work with the entire graph in main memory, otherwise, they would be prohibitively costly due to repeatedly random disk accesses. On the other hand, the vertex-centric processing is limited to problems that can be solved along the direct neighbors of the vertices (or with clever adaptations to such constraint); the good point is that it is well-suited to disk-based processing since it can suitably rely on sequential disk accesses. This kind of processing is not only prone to disk-based processing, but also to parallel processing according to which,

Algorithm 1 Vertex-centric graph processing

```
procedure GRAPH_SCAN(GRAPH G)
    for i = 1 to |V| do
        set_e ← set of edges adjacent to V[i]
        V[i].value ← f(set_e)
        for each edge e in set_e do
            e.value ← g(V[i].value, e.value)
```

Algorithm 2 Graph processing

```
procedure GRAPH_PROCESSING
    while convergence criterion is not satisfied do
        Graph_scan(G)
```

each thread can be responsible for a different share of the vertices. This possibility yields to quite effective algorithms.

2.2. Asynchronous parallel processing

Many researchers have developed systems to process graphs in large-scale, either using vertex-centric or edge-centric processing; this is the case of systems Pregel [23], Pegasus [17], PowerGraph [12], and GraphLab [22]. However, such systems are parallel-distributed, and thus, they demand knowledge, availability, and management of costly clusters of computers. More recently, a novel paradigm emerged in the form of frameworks that rely on asynchronous parallel processing, including systems GraphChi [19], TurboGraph [16], X-Stream [33] and MMap [21]. Such systems use disk I/O optimizations and the neighborhood information of nodes/edges in order to set up algorithms that can work in asynchronous parallel mode; that is, it is not required that their threads advance synchronously along the graph in order to reach useful computation. This approach has demonstrated success to tackle many problems, such as Pagerank, connected components, shortest path, and belief propagation, to name a few. In this paper, we use vertex-centric graph processing over framework GraphChi; however, our algorithm can be adapted to any of the frameworks available in the literature.

3. Related works

3.1. Clustering

The identification of lockstep behavior refers to the problem of partitioning both the rows and the columns of a matrix – known in the literature as co-clustering or bi-clustering. Some authors have worked on similar variations of the bi-clustering problem. For example, Papalexakis and Sidiropoulos used PARAFAC decomposition over

the ENRON e-mail corpus [29]; Dhillon *et al.* used information theory over word-document matrices [9], and Banerjee *et al.* used Bregman divergence for predicting missing values and for compression [3]. Other applications include gene-microarray analysis, intrusion detection [28], natural language processing [35], collaborative filtering [19], and image [8], speech and video analysis [13]. Note that, in this problem setting, whenever time is considered in the form of time windows to be detected, we have a Non-deterministic Polynomial-time (NP-hard) problem [2] that prevents the identification of the best solution even for small datasets. In fact, we deal with issues fundamentally different from the problems proposed so far, which, according to Kriegel *et al.* [18], are not straightly comparable due to their specificities.

Theoretically, our work resembles the works of Gupta and Ghosh [14] and of Crammer and Chechik [7]; similarly, we use local clustering principles, but, differently, we are not dealing with one-class problems. Besides, the core of our technique is a variant of mean-shift clustering [5], now considering temporal and multi-dimensional aspects. As we mentioned before, our contribution relates not only to performance, but also to a novel algorithmic approach.

3.2. Detection of suspicious behavior on the Web

One of the first algorithms tailored to detect suspicious behavior on the Web was designed by Douceur [10] in 2002. The author coins the term *sybil attack*, in the specific context of peer-to-peer networks. Sybil attacks are attacks in which a single entity can provide multiple identities, that is, a single node in the network can create or steal several other identities and use them to gain advantages, thus, undermining the security of the whole system. Latter, Newsome *et al.* [25] showed that sybil attacks can also occur in sensor networks where the attacker wants to bypass security measures, such as voting mechanisms and resource allocation policies.

One similar type of attack was studied by Chirita *et al.* [6] – the *shilling attack*; in shilling attacks, fake profiles are used to rate items in a recommendation system. Chirita *et al.* [6] proposed a technique to analyze profiles and to determine whether or not they are suspicious. Later, Su *et al.* [34] developed an algorithm to detect groups of shilling attacks, in which several profiles act in conjunction to alter the ratings of items in the system.

While both sybil and shilling attacks are similar to the concepts that we propose in this paper, as in defamation and illegitimate promotion, none of the aforementioned works consider the temporal dimension to detect the attacks. Time, in such setting, leads to a different problem with NP-Hard complexity [30]. Also, none of these works took into account the performance and scale that we consider in this paper.

Other related works use the graph theory to detect suspicious behavior on the Web. This is the case of algorithm Crochet [31] that aims at identifying quasi-cliques based on an innovative heuristic; it is also the case of MultiAspectForensics [24] that uses tensor decomposition to detect patterns within communities, including bipartite cores. In another work, Eigenspokes [32] uses singular-value decomposition to detect unexpected patterns in phone call data; also, Netprobe [27] uses belief propagation to find near bipartite cores in e-commerce graphs. Note, however, that: in spite of the many qualities of these related works, none of them focuses on performance at the same scale that we do; furthermore, they do not study the same problem that we do here, that is,

the detection of a set of users fraudulently interacting with the same set of products at around the same time. In fact, the closest approach to our work is the CopyCatch algorithm [4], which focuses on the unweighted version of the problem in a parallel, distributed setting – its experimental results reported used one *thousand* machines. In our work, we introduce a vertex-centric asynchronous parallel algorithm that runs in one *single* commodity machine, whose performance rivals to that reported in this former work, still achieving similar accuracy rates.

3.3. Lockstep formulation

In this section we provide a mathematical description of the lockstep detection problem. As it was mentioned in the introductory Section 1, we generalize the problem by amplifying its scope to defamation and illegitimate promotion. But before doing so, we present in the following the original formulation for the concept of a lockstep, which was formally defined by Beutel *et al.* [4] as a temporally-coherent near bipartite core. Along this section, please refer to Table 1 for a list of symbols and definitions.

Definition 1. A set of products P and a set of users U comprise an $[n, m, \Delta t, \rho]$-*temporally-coherent near bipartite core* if and only if there exists $P_i \subset P$ for all $i \in U$ such that:

$$|P| \geq m \tag{1}$$

$$|U| \geq n \tag{2}$$

$$|P_i| \geq \rho |P| \ \forall i \in U \tag{3}$$

$$(i, j) \in E \ \forall i \in U, j \in P_i \tag{4}$$

$$\exists t_j \in \mathbb{R} \ s.t. \ |t_j - L_{i,j}| \leq \Delta t \ \forall i \in U, j \in P_i \tag{5}$$

In other words, we have a lockstep if we find a set of products P that was recommended by a set of users U within a Δt time window; we relax this definition with parameter ρ, which states that we also have a lockstep if we partially (ρ percentage) satisfy this definition. Note that what makes the problem even more challenging is the temporal factor; also, note that the problem refers to reducing the search space of frauds by pointing out suspicious behaviors, which can turn out to be actually fraudulent, or not. Figure 1 illustrates the concept of a lockstep. It shows how bipartite cores are formed and it also highlights the independence of the time-windows that are exclusive to each product.

While we are considering the aforementioned definition of suspiciousness, it is also important to discuss how effective it is in preventing malicious agents from manipulating recommendations. In other words, we are interested in finding how much damage agents could inflict without being detected. The core fact in the definition of an attack is that: the smaller the attack, the smaller its harm; in consequence, while it is hard to detect very small attacks, they tend to have no use unless they occur in extremely high cardinality. The boundaries of this relation for a non-temporal version of the problem are an open problem, as it is discussed in previous works [4, 36]. The challenge becomes even harder when time is considered, as we do in this paper, which sets up an extension of the problem.

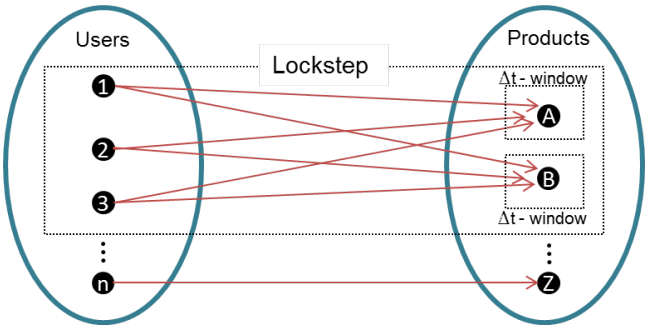


Figure 1: Lockstep illustration: A group of users (1,2 and 3) recommends a group of products (A and B), within limited time-windows for each product, forming a bipartite core.

In this context, it is possible to conclude that, while we may miss very small attacks and trying to detect them might raise the number of false positives, given the controlled conditions, an adversary can only do a limited amount of damage without getting caught. Finding an optimal strategy of attack is also related to the same open problem discussed before, and since we consider individual temporal centers for each lockstep, one cannot merely use common strategies, such as to wait for a given amount of time before attacking other products with the same users as he/she would be caught anyway. Additionally, the fine tuning of our algorithm's parameters allows the user to detect different types of attacks, further diminishing the possibilities of an adversary to bypassing the system.

4. Methodology

4.1. The generalized lockstep problem

This section shows how to enhance the potential semantics of the lockstep-detection problem by taking into account the weights of edges (e.g., recommendations' scores).

Given the formulation presented in Section 3.3, we propose new semantics to the problem by considering weights of edges to define the concepts of defamation – Equation 6 – and illegitimate promotion – Equation 7. These weights correspond, for instance, to the numeric evaluation (score) given by a user to a product in a recommendation website. Our formulation considers the weights to be positive integers, and we use a threshold κ to distinguish between defamation and promotion.

$$W_{i,j} \leq \kappa \ , \ i \in U, j \in P_i \tag{6}$$

$$W_{i,j} \geq \kappa \ , \ i \in U, j \in P_i \tag{7}$$

We consider the problem as an optimization problem, whose objective is to catch as many suspect users as possible, while only growing P until parameter m is satisfied.

Symbol	Definition
M and N	Number of nodes in each side of the bipartite graph.
C	Set of locksteps.
I	$M \times N$ adjacency matrix.
L	$M \times N$ matrix holding the timestamp of each edge.
W	$M \times N$ matrix holding the weight of each edge.
$U[c]$ and $P[c]$	Set of users or products in lockstep c.
m and n	Minimum number of products and users in the lockstep to be considered valid.
Δt	Size of the timespan.
ρ	Threshold percentage that the cardinality of the sets of products and users must satisfy to be in a lockstep.
nSeed	Number of starting seeds for the algorithm to begin searching for locksteps.
λ and κ	Function and threshold used to define defamation and promotion.
v_j	Current average time of suspicious recommendations to product j.

Table 1: Symbols and Definitions.

Our objective function is in Equation 8. The goal is to find $U[c]$ and $P[c]$ to maximize the number of users and their interactions for a given cluster c.

$$\underset{U[c],P[c]}{\arg\max} \sum_i q(L_{i,*}|c, W_{i,*}|c, P[c]) \tag{8}$$

where

$$q(u,w,P[c]) = \begin{cases} \sigma \text{ if } \sigma = \sum_{j \in P[c]} I_{i,j} \phi(v_j, u_j) \lambda(w_j) \geq \rho m \\ 0 \text{ otherwise} \end{cases} \tag{9}$$

$$\phi(t_v, t_u) = \begin{cases} 1 \text{ if } |t_v - t_u| \leq \Delta t \\ 0 \text{ otherwise} \end{cases} \tag{10}$$

$$\lambda(g_j) = \begin{cases} 1 \text{ if } g_j \geq \kappa \\ 0 \text{ otherwise} \end{cases} \quad \text{for promotion} \tag{11}$$

$$\lambda(g_j) = \begin{cases} 1 \text{ if } g_j \leq \kappa \\ 0 \text{ otherwise} \end{cases} \quad \text{for defamation} \tag{12}$$

Equations 11 and 12 refer to our definitions of illegitimate promotion and defamation, respectively, while Equation 9 shows how we incorporate these weight constraints in the original problem, through the definition of a threshold function λ. That is, we expand the formulation by including new information relative to the weight of these relationships, as well as incorporate such definitions in the objective function, effectively broadening the scope of the problem and its potential applications.

4.2. Algorithm ORFEL

In order to find locksteps, this section presents the **Online-Recommendation Fraud ExcLuder (ORFEL)**, a novel, iterative algorithm that leverages the idea of

vertex-centric processing introduced in Section 2.1 to expand and improve both the scope (weighted graphs) and the efficiency (scalability on a single computer) of current state-of-the-art approaches. Each iteration of our algorithm executes two functions: $updateProducts$ and $updateUsers$ that, respectively, will add/remove products and users from a lockstep that is being identified. The algorithm iterates until convergence – that is, until sets P and U stabilize for all the locksteps that were found; we consider that a lockstep is stable when no new product or user enters or leaves the lockstep in one iteration, compared to the previous one. The full pseudo-code of ORFEL is in Algorithm 3.

Initialization

The algorithm relies on seeds to search the data space; the general idea is to have each seed inspecting its surroundings looking for one local maximum. Each seed in the algorithm corresponds to one potential lockstep, which comprises a set of products and a set of users. The initial seeds correspond to minimum locksteps, that is, locksteps with one single product, and a few (≥ 1) users each – the only requirement for the initial set of users is that it cannot be an empty set.

The initialization step randomly chooses products of the dataset, each one corresponding to one seed. Then, for each product (seed), ORFEL forms initial locksteps by randomly choosing a constant, small number of users that recommended this product. This is necessary so that the algorithm has initial elements – $P[i]$ and $U[i]$ for every ith-lockstep – scattered throughout the search space. Later, the initial locksteps will grow iteratively in number of products and users.

Product update

In procedure $updateProducts$ – see Algorithm 4 – we only consider vertices that are *products*, so modifications occur in set $P[i]$ only. This function is called for every product to test if it fits in one of the locksteps; the test is performed for all locksteps. One product enters a given lockstep if at least ρ percent of the users currently in the lockstep recommended that *product* within a Δt time window. To compute this percentage, the algorithm only considers recommendations that fit the given weight constraint (represented by the λ function), which characterizes either defamation or illegitimate promotion.

For locksteps with m products, that is, those with the maximum number of products, we test if it is worth to swap one of its products for the candidate one. This test is similar to the one used to add a product, except that, to be swapped, now the candidate product must contain a superset of the set of recommendations that the current product has. This is a heuristic approach that leads to an additional coverage of the search space because, as we look for supersets of recommendations, the locksteps tend to increase in size.

User update

Procedure $updateUsers$ – see Algorithm 5 – considers only vertices that are *users*, so it modifies set $U[i]$ only. Similarly to what is done in step $updateProducts$, we update each lockstep separately by testing if the current *user* can be added to it. A candidate *user* will enter a lockstep if it recommends at least ρ percent of the products in the

cluster within a $2\delta t$ time window of each of the products' time centers - that is, the average recommendation time on that product inside the lockstep - and if it fits the desired weight constraint. If the candidate *user* fills the requirements, it is added to that lockstep. Note that this step allows *users* outside the actual δt time window to enter the lockstep; this is the mechanism that drives the lockstep towards a better local maximum, whenever there exists one. We propose to use $2\delta t$, following empirical evidence obtained from both our work and the state-of-the-art Beutel's [4] approach.

End iteration

As it can be seen in Algorithm 3, we run procedure *endIteration* right after step *updateUsers* is complete. This additional step is described in Algorithm 6. For all locksteps, the algorithm sorts the $2\Delta t$ recommendations by their timestamps and scans them sequentially looking for the subset that maximizes the recommendation criterion (number of recommendations); the target subset must fit a Δt window. This is the core mechanism of our algorithm; what it does is to let a $2\Delta t$ time window to take place at first, then, from the corresponding set of recommendations, it selects a subset that maximizes the target criterion. This mechanism is what makes the seeds "inspect" their $2\Delta t$ neighborhoods. If the recommendation set changes, a new iteration will lend new products and users to entering/swapping into the lockstep, until convergence. Once a seed finds a local maximum, it stops evolving and does not change anymore.

Note that some seeds may converge sooner than others, leading to locksteps smaller than parameters m and n. These seeds are considered "dead" (no modifications between iterations), so they are ignored by the algorithm. They can occur from the second iteration on, after which the number of locksteps (live seeds) becomes smaller than the initial number of seeds. The algorithm converges when all seeds are "dead".

Algorithm 3 ORFEL Algorithm.

```
function ORFEL(n, m, ρ, Δt, nSeeds)
    Initialize U[nSeeds], P[nSeeds]                ▷ Initial Seeding
    repeat
        U' = U
        P' = P
        for each product p in |V| do
            P = updateProducts(p)
        for each user u in |V| do
            U = updateUsers(u)
        endIteration()
    until U' = U and P' = P
    return [U, P]
```

4.3. Discussion about the parameters

As it can be seen in Algorithm 3, *ORFEL* has five parameters: m, n, ρ, Δt and *nSeeds*. The first two, m and n, respectively refer to the cardinality of *products* and *users* that

Algorithm 4 updateProducts

```
procedure UPDATEPRODUCTS(vertex)
    for each Lockstep c ∈ C do
        Recomms ← U[c].edges ∩ vertex.edges
        timeCenter ← avgtime(Recomms)
        for each edge e in Recomms do
            if |e.time − timeCenter| > Δt and λ(e.weight) then
                Recomms = Recomms − {e}
        if |P[c]| < m then
            if (|Recomms|/|U[c]|) ≥ ρ then
                P[c] = P[c] ∪ {vertex}
        else
            for each product p ∈ P[c] do
                if p.Recomms ⊂ Recomms then
                    swap = p
            P[c] = (P[c] − {swap}) ∪ {vertex}
```

Algorithm 5 updateUsers

```
procedure UPDATEUSERS(vertex)
    for each Lockstep c ∈ C do
        Recomms ← P[c].edges ∩ vertex.edges
        for each edge e in Recomms do
            pCenter ← avgtime((u, e.vertex), u ∈ U[c])
            if |e.time − pCenter| > Δt and λ(e.weight) then
                Recomms = Recomms − {e}
        if (|Recomms|/|P[c]|) ≥ ρ then
            U[c] = U[c] ∪ {vertex}
```

Algorithm 6 endIteration

```
procedure ENDITERATION
    for each Cluster c ∈ C do
        for each product p ∈ P[c] do
            Sort U[c] by the time of the Recomms
            Scan sorted U[c] for the 2Δt-subset that maximizes the number of
Recomms
            Remove the users from U[c] that are not in the subset
```

the algorithm verifies when evaluating suspicious locksteps. Parameter ρ is the minimum percentage (the tolerance fraction) of products $\rho * m$ for the algorithm to state that a bipartition is, in fact, suspicious. Although one can freely alter the value of ρ, based on empirical evidence, we suggest using no less than 80%, otherwise, the locksteps might degenerate. Note that we use a single value of ρ for both users and products, because, intuitively, this parameter is expected to be nearly the same for the two entities; nevertheless, algorithmically, one could easily adapt our proposal to use different values at the cost of greater computational complexity. Parameter Δt defines the time window within which the interactions (recommendations) should take place. Finally, parameter *nSeeds* refers to the number of seeds that the algorithm will spread through the search space, each one looking for one suspicious lockstep.

Parameters m and n define the aspects of the suspicious behaviors that we are looking for. Increasing (or decreasing) the value of m or n means that we want to find suspicious behaviors involving more (or less) *products* and/or *users*. These values define what we call "*AttackSize*", i.e., the dimensions of the attacks that we presume to exist. In practice, parameters m and n filter out attacks that are too small and/or too large, what may be desired depending on the domain. In the experimental Section 5, we evaluate how distinct configurations of *AttackSize* impact the efficacy of our algorithm.

Parameter ρ makes the algorithm flexible about different types of attacks, including those in which the *users* attack only a fraction (percentage) of the expected number of products m in the locksteps. The value of ρ defines how tolerant we want to be with respect to the very definition of suspiciousness. If we set ρ to 1, only perfect full locksteps would be considered, in which every user recommended every product of the cluster. On the other hand, if we set ρ too low, such as $\rho = 0.5$ for instance, only half of the users would have to recommend each product, possibly leading to incorrect assumptions about the concept of suspiciousness. In practice, ρ defines that the algorithm should have a tolerance around m and n. Note that parameters ρ, m and n depend on the semantics of the problem's domain and ought to be different for each application. It is also true for parameter Δt, which we describe as follows.

Parameter Δt is the time span to be defined by the analyst when searching for attacks. For instance, let us assume the context of attacks in a social network; in this setting, one could argue that a time span as large as a couple of hours is enough to find ill-intended interactions. On the other hand, in the context of online reviews, the time span of one week could be more appropriate. Note that the same reasoning can also be used to define parameters m and n.

Finally, parameter *nSeeds* controls *ORFEL*'s potential of discovery; as we show in the experiments (see Figure 3), the minimum number of seeds required to analyze one given dataset follows a linear correlation with the data size.

4.4. Convergence

Our algorithm finds a set of local maxima for the objective function defined in Equation 8. Note that this function is bounded, since the sets of users and products are limited. Therefore, convergence depends solely on the behavior of steps $updateProducts$, $updateUsers$ and $endIteration$.

In step $updateProducts$, the algorithm checks if a given product should be added to any of the current locksteps, deciding to include or to swap that product only if it

covers more recommendations than what we have so far. As a result, the objective function may only improve or stay unaltered after this step. On the other hand, step $updateUsers$ attempts to add suspect users to the existing locksteps by extending the size of the time-window, while step $endIteration$ makes sure that only the largest set of users fitting in the best Δt time-window is added to each lockstep. As a consequence, these last two steps can only improve our objective function, by including more users, or leave it unaltered if no users are added.

These observations lead us to conclude that the locksteps grow asymptotically in our algorithm, eventually reaching a local maximum that prevents changes between two iterations. Therefore, $ORFEL$ always converges despite the data given as input. Besides this theoretical exercise, the convergence of our algorithm is empirically demonstrated in Section 5.

4.5. Computational cost

To study the computational cost of our algorithm, let us assume that the graph G received as input has size D bytes; we have M bytes available in main memory, and; the disk blocks have b bytes each. In this setting, $ORFEL$ splits the graph into $\lceil P = D/M \rceil$ parts. Each part contains edges that are sorted in disk according to their source vertices so that the graph is processed by reading the parts twice, first as targets and then as sources. Therefore, in order to read the entire graph, it is necessary to read $B = D/b$ disk blocks twice, or $2B$ times. For each part it is also necessary to read the other $P-1$ parts, leading to P^2 disk seeks. Therefore, the cost of disk operations is given by P^2 *disk seeks* $+2B$ *block reads* per iteration.

$ORFEL$ runs for I iterations. In each iteration, besides the disk operations, it runs once for each of the S seeds (worst case) processing in memory all the $|E|$ edges of the graph at each time. Therefore, the processing cost of the algorithm is $I * O(S * |E|)$.

Each iteration of the algorithm asks for a reorganization step in which the locksteps of each seed are redefined based on the results annotated in the last iteration. For I iterations, S seeds, n users and m products, this step runs at cost $I * O(S * n * (m * log(m)))$. Part of this cost is due to the operation of sorting in memory (logarithmic time). This is the worst case scenario, when the algorithm processes all seeds – the cost drops abruptly after a few iterations because the majority of the seeds does not grow; instead, they stop evolving at a local maximum that is too small to be considered a lockstep, being ignored in further iterations.

Finally, the total cost of $ORFEL$ is $I * (P^2$ *disk seeks* $+2B$ *block reads* $+ O(S * |E|) + O(S * n * m * log(m)))$. Note that the cost of processing is irrelevant, since it is 6 orders of magnitude smaller than that of a mechanical disk and 4 orders smaller than that of a solid-state disk. As so, the main cost of $ORFEL$ is $I * (P^2$ *disk seeks* $+$ $2B$ *block reads*$)$. Note, from our analysis, that the computational cost depends on the amount of main memory available, which is used as a buffer for data coming from disk; hence, all the runtime measurements reported in the next section could be smaller if we had more memory to use.

Dataset	# Users	#Products	# Total nodes	# Edges
Amazon.FineFoods	256,059	74,258	330,317	568,454
Amazon.Movies	889,176	253,059	1,142,235	7,911,684
Synthetic.C	2,000,000	8,000,000	10,000,000	100,000,000

Table 2: Datasets.

5. Experiments

5.1. Experimental setting

We implemented *ORFEL* using Java 1.7 over the GraphChi platform, as stated in Section 2.2. We ran our experiments on an i7-4770 machine with 16 GB of RAM, and 2TB 7200RPM HDD; for the tests with SSD, we used a 240GB drive with I/O at 450MB/s. For full reproducibility, the complete experimental setup is publicly available at `www.icmc.usp.br/pessoas/junio/ORFEL/index.htm`, including source codes and graph/lockstep generators.

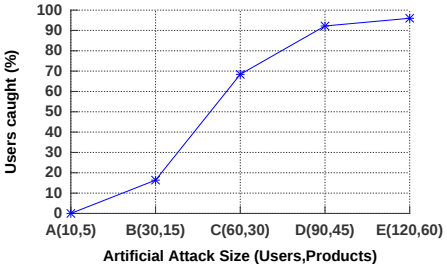
We studied two real-world graphs: *Amazon.FineFoods* and *Amazon.Movies*. They are publicly available at the Snap project [20] web page at `snap.stanford.edu/`. Both datasets comprise user-product recommendation data from the Amazon website; the first one refers to the section of fine food products and the second one contains reviews of movies. For each review, we have the corresponding timestamp and one numeric evaluation (score) ranging from 1 to 5. Synthetic graphs were also studied so to generalize the scope of our tests. To generate the data, we used a bipartite graph generator that works based on the G_{nmk} model available on NetworkX [15], in which n stands for the number of nodes in the first bipartite set; m stands for the number of nodes in the second bipartite set; and k is the number of randomly generated edges connecting both sets. Table 2 lists the two Amazon datasets and the synthetic dataset *Synthetic.C*, which was generated using $n = 2,000,000$, $m = 8,000,000$ and $k = 100,000,000$. Additionally, we generated benchmark datasets that are larger versions of dataset *Synthetic.C*; they were used to study the scalability of our algorithm, as it is described in Section 5.4.

Experimental goals

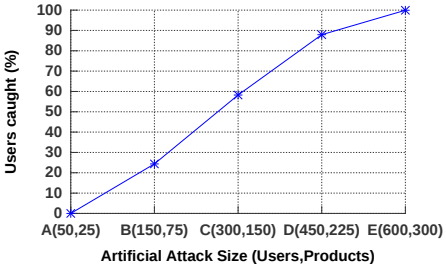
The main feature expected from *ORFEL* is the ability to detect lockstep attacks, either those related to defamation or the ones of illegitimate promotion. As it was mentioned in Section 3.1, this is one NP-hard problem that we approximately solve via an optimization approach. Considering these aspects, we verify: the correctness of our algorithm in Section 5.2; its efficacy (i.e., the ability to find the majority $> 95\%$ of the lockstep attacks) in Section 5.3; and; its efficiency (i.e., the ability to reach efficacy within desired time constraints) in Section 5.4.

5.2. Preliminar tests under controlled conditions

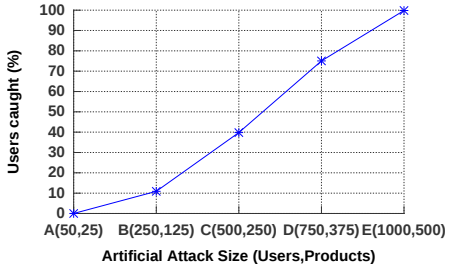
In the first experiment, we used 4 small (thousand-edge scale) synthetic graphs to verify if the algorithm detects locksteps only when they really exist. These are the controlled conditions of our experimentation, that is, we wanted to make sure that the



(a) Amazon.FineFoods - Parameters (10,5,0.8,1000)



(b) Amazon.Movies - Parameters (50,25,0.8,1000)



(c) Synthetic.C - Parameters (50,25,0.8,3000)

Figure 2: Experiments of efficacy: the percentage of attacks caught versus the size of the artificially generated attacks. Parameters are described as (n,m,ρ,nSeeds).

algorithm would not point out suspect behaviors when we had ascertained that there were none to be detected. We generated synthetic graphs in which no product, nor set of products would configure a given suspect behavior, such as 10 products and 25 users. We then ran *ORFEL* with varying parameters, including $m = 5$, $n = 10$ and $m = 10$, $n = 25$ and verified that, as expected, no lockstep was detected. The same results could be inferred from the algorithm description in Section 4.2 and also from the discussion in Section 4.4, still, we verified this feature empirically.

5.3. Efficacy

We define efficacy as the ability to identify the majority ($> 95\%$) of the locksteps. To test this feature, we created controlled conditions with artificial attacks appended to our datasets that allowed us to evaluate the output of the algorithm. Algorithm 7 shows how we generated such attacks, by randomly choosing a group of products and users and then connecting them within a limited Δt time window. This was necessary because, since the problem is NP-hard, we would not be able to know whether or not the output of the algorithm is correct considering uncontrolled conditions. This problem is a variation of subspace clustering, considering the semantic that the clusters (locksteps) are unusual and, therefore, suspect. Note that we did not focus on the issue of determining whether or not a given suspect lockstep is actually an attack, since this is one distinct problem that demands extra information (i.e., identification, customer

profile, and so on) to be evaluated by means of false-positive and true-positive rates.

Algorithm 7 Lockstep Generator

```
procedure LOCKSTEPPER(Graph G, nUsers, nPages, Δt)
    users = GetRandomUsers(G, nUsers);
    pages = GetRandomPages(G, nPages);
    for each Page P ∈ Pages do
        timestamp = getRandomTimeStamp();
        rating = getRandomRating();
        for each User U ∈ Users do
            newTimeStamp = timestamp + getRandomVariation(Δt);
            addEdge(G, U, P, newTimestamp, rating);
```

Attack size

In order to analyze the ability of the algorithm to find locksteps of different sizes, we ran experiments for each of the three datasets described in Table 2. We fixed m and n in each case while varying the sizes of the artificial attacks appended to the dataset, so to be able to see how effective the algorithm is, depending on the size of the attacks.

In the first experiment, for each dataset of Table 2, we appended artificial attacks to the data with sizes varying from 10 users and 5 products to 1,000 users and 500 products. This allowed us to observe the percentage of attacks caught for each configuration – in Figure 2(a), ($n = 10$,$m = 5$,$\rho = 0.8$,$nSeeds = 1000$); in Figure 2(b), ($n = 50$,$m = 25$,$\rho = 0.8$,$nSeeds = 1000$); and, in Figure 2(c), ($n = 50$,$m = 25$,$\rho = 0.8$,$nSeeds = 3000$). One can see a similar behavior in all plots; that is, the percentage of users caught tends to grow as their sizes become larger than the size described by the input parameters given to the algorithm. Intuitively, the larger the attacks are, the more likely that they will be detected using a given configuration. Concomitantly, the smaller the attacks, the less harmful they are. Lastly, notice that even the smaller attacks could be detected with proper parameters – in this experiment, however, we wanted to demonstrate the general behavior of the algorithm when using specific parameter settings, and not whether or not smaller attacks could be detected.

This experiment also indicates that ORFEL behaves as expected for such task in terms of efficacy. That is, if we compare the behavior of ORFEL with that of the state-of-the-art algorithm CopyCatch – see Figure 6b at [4] – one can see that both approaches present a very similar curve for spotting artificial attacks according to the attack size and the algorithm parameterization. The results in [4] maintain the same intuition regarding which attacks are easier to detect and how important is the parameter tuning, therefore, they corroborate our conclusions with regard to ORFEL's efficacy.

Number of seeds

We also used our three datasets to study the behavior of $ORFEL$ regarding the number of seeds that it uses. We ran each experiment 4 times in each dataset – as the algorithm is non-deterministic – and report the average response. It was a requirement that none

of the 4 runs would present discrepant results, and we were able to verify this desirable property since the variance of the results was on the order of 1%. We introduced 20 synthetic attacks (10 defamations and 10 illegitimate promotions) in each dataset and varied the number of seeds from 1,000 to 7,000. Figure 3 reports the results obtained in these experiments. For dataset Amazon.FineFoods – the smaller one with 550 K edges – we were able to catch over 95% of the attacks with 4,000 seeds. Interestingly, the average of attacks caught with 5,000 seeds was significantly lower, indicating that the algorithm reached its peak performance with nearly 4,000 seeds and only had some variation afterward due to its non-determinism. Figure 3 also reports that the algorithm caught over 95% of the attacks with 6,000 seeds in dataset Amazon.Movies (8 M edges), and; it caught over 95% of the attacks with 7,000 seeds in dataset Synthetic.C (100 M edges). These results indicate that the best number of seeds follows a linear correlation with the data size, being approximately $10^3 * log(number\ of\ edges)$. For our 3 datasets, it is $\sim$ 5800, $\sim$ 6900 and $\sim$ 8000 respectively.

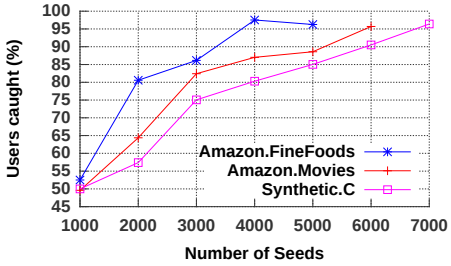


Figure 3: Experiments of efficacy: the percentage of attacks caught versus the number of seeds. Efficacy is demonstrated when over 95% of the attacks are caught. Parameters [n,m,ρ,AttackSize(Users,Products)] are: Synthetic.C [50,25,0.8,(750,375)]; Amazon.Movies [50,25,0.8,(500,250)]; Amazon.FineFoods [10,5,0.8,(50,25)].

From this experiment we verified that *ORFEL* is effective; it identified more than 95% of the attacks in three datasets of different sizes. Also, notice that the algorithm accurately detected attacks of distinct sizes, as it can be seen in the parameters of Figure 3. It means that ORFEL fits the peculiarities of distinct domains.

Experiments on real data

We performed additional experiments using our real datasets Amazon.Movies and Amazon.FineFoods, this time *without including any synthetic data*. In this context, we considered that a suspect behavior would be 20 users positively recommending 6 movies or food products in less than a week, which, in this semantic context, is an intense load of recommendations. We ran the algorithm and found 37 suspect locksteps; it took 8 minutes to achieve convergence for the largest dataset, Amazon.Movies. Since the execution time was quite small, we also tested the algorithm using variations of the initial attack description, with 15 users and 7 movies within a week, and also 10 users and 10 movies within three days. After manually analyzing the suspect locksteps, we discovered that they were caused by amazon's policy of using different identification numbers for different flavors/sizes of the same food product, and for different versions

of the same movie, while merging their reviews. Although the locksteps found were not actual attacks, this simple experiment revealed a behavior that should be better analyzed since the aforementioned Amazon's policy could eventually lead customers to misleading choices. In summary, we were able to identify, in a universe of 1,140,000 nodes and 8,000,000 edges, tiny temporal patterns that demand close attention to be noticed. We also emphasize the reduced time required to obtain such results, which allowed us to study different parameter settings according to the data semantics.

5.4. Efficiency and scalability

Due to the current scale of network-like data, our method must be efficient. That is, it must handle billion-scale graphs in reasonable time. We tested this requirement with synthetic, benchmark datasets that are larger versions of dataset $Synthetic.C$. Although there are plenty of real data related to our problem (network data, including edge weights and time stamps), such data is rarely shared by companies due to privacy matters.

Preprocessing

Asynchronous Parallel Processing platforms like those reviewed in Section 2.2 demand a preprocessing step in which the data is organized and formatted in accordance to the platform's paradigm. In our case, this step converts text to binary data, then it sorts and writes the vertices in order, so to have them read from disk with sequential scans, minimizing the number of seeks. We take nearly 45 minutes, wall-clock time, to preprocess 1 billion edges on a mechanical disk, and nearly 15 min. on a solid-state disk – for a given dataset, preprocessing is necessary only once, no matter how many times we shall process the data later on.

Number of edges

We tested the time scalability of our algorithm regarding the number in edges of the input graph. In the first experiment, we ran $ORFEL$ with 100 seeds; we took 7 runtime measurements with the number of edges varying from 50 million to 1 billion, each of these measures were obtained as the average of 3 individual runs. Figure 4 reports the results for the mechanical disk; clearly, the runtime scales linearly with regard to the number of edges. For this configuration, $ORFEL$ took 143 min. ($\approx$2.38 hour) to process 1 billion edges stored on a mechanical disk, and 78 min. ($\approx$1.3 hour) using a solid-state disk.

We argue that this performance is *very efficient* because the previous work (see Figure 4a in Beutel et al. [4]) took $\approx$0.5 hour to do a similar processing with *one thousand* machines over MapReduce, while we used one *single* commodity machine. Our gain in performance is considerable because the former work executes a sequential (non-parallel) algorithm to compute one seed at a time in each machine; therefore, performance comes at the cost of using thousands of machines, each one executing an instance of the computation, in a distributed environment that has heavy communication demands. Differently, our algorithm explores the fact that the problem can be solved considering only the neighborhood of each node, thus allowing us to process the graph in a parallel asynchronous mode with multiple seeds

being processed simultaneously. Note that our gains in performance were not only remarkable – they also made it possible to tackle the problem using commodity hardware.

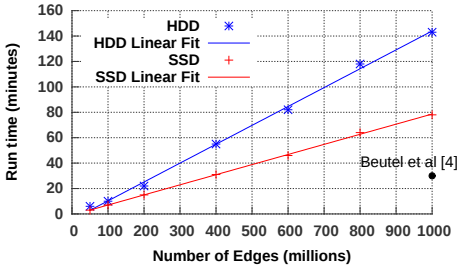


Figure 4: Experiments of scalability on the number of edges: linear growth of computation time (100 seeds) versus the number of edges for mechanical disk (HDD) and for solid-state disk (SSD). Parameters [n,m,ρ,AttackSize(Users,Products),nSeeds] are: [50,25,0.8,(500,250),100].

Number of seeds

We also studied the runtime of *ORFEL* with distinct numbers of seeds. In this experiment, we used a graph with 100 million edges varying the number of seeds from 100 to 5,000. Figure 5 reports the results; as it can be seen, our proposed method scaled linearly in time with regard to the number of seeds. The algorithm took 10 min. to process the data using 100 seeds, while it took 298 min. with 5,000 seeds; that is a 30-times increase in runtime for a 50-times increase in the problem input.

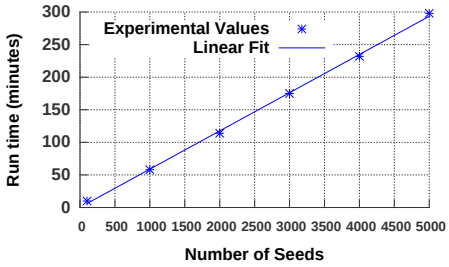


Figure 5: Experiments of scalability on the number of seeds: linear growth (line coefficient $<$ 1) of computation time versus the number of seeds for mechanical disk (HDD) over 100 million edges. Parameters [n,m,ρ,AttackSize(Users,Products)] are: [50,25,0.8,(500,250)].

6. Conclusions and future work

6.1. Conclusions

We conclude that, although the problem of detecting fake online interaction over time is NP-hard, it is possible to timely detect most of the malicious activities even with low-cost computer machinery. To do so, we designed a vertex-centric-based graph

algorithm using the asynchronous parallel processing paradigm. The general problem is modeled as a bipartite weighted and timestamped graph from which we want to detect temporal near-bipartite cores. This model suits to problems of systematic attacks aimed at defaming or promoting online entities in applications of high-impact, e.g., user-product recommendations, user-app evaluations and journal-journal co-citations, which may be performed by means of fake users, malware credential stealing, Web robots, and/or social engineering. To validate our proposal, we studied two real graphs of e-commerce, besides synthetic data.

Finally, we emphasize the importance of detecting lockstep behavior, either for defamation or promotion, because these frauds may harm both customers and vendors by inducing sales of unverified products. We also note that this problem gets even more relevant as the Web 2.0 expands, in which the habits of the users are heavily influenced by online trading and recommendation.

6.2. Future work

First, an interesting direction would be further analyzing the boundaries of potential damage an attacker could inflict without being detected, given by the compromise between the size of the attack and the number of attacks. Which as stated in 3.3 is an open problem. Also, as we mentioned before in Section 1, *ORFEL* suits other problems that can be represented as graphs, lending support to additional applications as detailed next.

6.2.1. Social networks

In social networks, the usual interaction is to like a given post, such as in Google+ or in Facebook; for this configuration, locksteps characterize solely illegitimate promotion, in which a given post (or page) gets fake likes from attackers willing to make it more relevant than it really is. According to the model of *ORFEL*, this problem refers to one unweighted bipartite graph, i.e., all edges weight the same.

6.2.2. Journal co-citations

Given the pressure for relevance and impact, some scientific journals may use a co-citation scheme in which one journal cites the other and vice-versa, just like in the case spotted by Nature in 2013 [26]. According to this scheme, which is one variant of the lockstep behavior, a journal tends to favor papers that cite a specific journal; editors may even recommend authors what to cite in their work as a condition for publication.

To identify this kind of lockstep behavior is not a trivial task because systematic co-citation tends to "disappear" along years of publications, provided that such schemes are usually covered by the volume of legitimate citations and by the magnitude of time. For example, it is reasonable to have co-citation between any two journals in a period of 10 years. The problem becomes even harder if more than two journals – e.g., three or four journals – set up the scheme. In this case, a simple journal-to-journal interaction may not be sufficient to detect the scheme. The temporal factor and the volume of data make it a problem much harder than simply detecting bipartite subgraphs.

This problem is another instance of the lockstep detection problem studied in our work. With *ORFEL*, it is possible to spot co-citation occurring, let us say, within

periods of 1 or 2 years for any number of journals. For time intervals such as those, one may suspect if a set of journals cite each other with high intensity.

In this specific case, our problem formulation changes a little. Our model assumes users recommending products – a bipartite graph; for detecting journal co-citations we must replicate the set of journals under investigation. That is, each journal must be represented twice in the model: once as a citing journal and one other time as a cited journal, thus, defining a bipartite graph as expected by our algorithm. The output of *ORFEL*, then, shall present bipartite subgraphs. However, distinctly from the user-product model, it is not enough to identify bipartite subgraphs as an indication of fraud; we must also have a high similarity between the two sets of nodes in each subgraph. This similarity can be straightly evaluated using the Jaccard set similarity: $Jaccard = |set_1 \cap set_2|/|set_1 \cup set_2|$, which returns 1 if two sets are exactly the same, and 0 if they have no intersection. For the co-citation problem, our algorithm could be configured to return the set of bipartite subgraphs ordered by their Jaccard similarity. Of course, *ORFEL* spots behaviors that are solely suspicious – not definitive frauds; they must go through human interpretation for a definitive decision, considering, for example, that it is expected that the journals with very high impact rates cite each other, while the same behavior is not expected for journals with lower impact rates.

Note that our algorithm cannot only detect suspicious co-citation cases – it can do it very efficiently. Since *ORFEL* is fast and scalable, it can virtually inspect <u>all</u> the publication interaction ever produced in just a few hours.

7. Acknowledgments

We thank Prof. Christos Faloutsos and Alex Beutel, from Carnegie Mellon University, for their valuable collaboration. This work was funded by Conselho Nacional de Desenvolvimento Cientifico e Tecnologico (CNPq - 444985/2014-0), Fundacao de Amparo a Pesquisa do Estado de Sao Paulo (FAPESP - 2013/10026-7, 2016/02557-0 and 2014/21483-2), and Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior (Capes).

References

[1] Akoglu, L., McGlohon, M., and Faloutsos, C. (2008). Rtm: Laws and a recursive generator for weighted time-evolving graphs. In *Data Mining, 2008. Eighth IEEE International Conference on*, pages 701–706. IEEE.

[2] Anagnostopoulos, A., Dasgupta, A., and Kumar, R. (2008). Approximation algorithms for co-clustering. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Database Systems*, pages 201–210. ACM.

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

[4] Beutel, A., Xu, W., Guruswami, V., Palow, C., and Faloutsos, C. (2013). Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*, pages 119–130.

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

[6] Chirita, P.-A., Nejdl, W., and Zamfir, C. (2005). Preventing shilling attacks in online recommender systems. In *Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 67–74. ACM.

[7] Crammer, K. and Chechik, G. (2004). A needle in a haystack: local one-class optimization. In *Proceedings of the twenty-first international conference on Machine learning*, page 26. ACM.

[8] Cruz-Roa, A., Caicedo, J. C., and González, F. A. (2011). Visual pattern mining in histology image collections using bag of features. *Artificial intelligence in medicine*, 52(2):91–106.

[9] Dhillon, I. S., Mallela, S., and Modha, D. S. (2003). Information-theoretic coclustering. In *Proceedings of the ninth ACM international conference on Knowledge discovery and data mining*, pages 89–98. ACM.

[10] Douceur, J. R. (2002). The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer.

[11] Facebook (2012). Improvements to our site integrity systems. `http://facebook.com/10151005934870766`. october, 2014.

[12] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). Power-Graph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30.

[13] Goyal, A., Ren, R., and Jose, J. M. (2010). Feature subspace selection for efficient video retrieval. In *International Conference on Multimedia Modeling*, pages 725–730. Springer.

[14] Gupta, G. and Ghosh, J. (2005). Robust one-class clustering using hybrid global and local search. In *Proceedings of the 22nd international conference on Machine learning*, pages 273–280. ACM.

[15] Hagberg, A. A., Schult, D. A., and Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, CA USA.

[16] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H. (2013). TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the 19th ACM international conference on Knowledge discovery and data mining*, KDD '13, pages 77–85, New York, NY, USA. ACM.

[17] Kang, U., Tsourakakis, C. E., and Faloutsos, C. (2009). PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE.

[18] Kriegel, H.-P., Kröger, P., and Zimek, A. (2009). Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Transactions on Knowledge Discovery from Data*, 3(1):1.

[19] Kyrola, A., Blelloch, G., and Guestrin, C. (2012). GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 31–46.

[20] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[21] Lin, Z., Kahng, M., Sabrin, K. M., Chau, D. H. P., Lee, H., and Kang, U. (2014). Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *Big Data, 2014 IEEE International Conference on*, pages 159–164. IEEE.

[22] Low, Y., Gonzalez, J., and Kyrola, A. (2010). Graphlab: A new framework for parallel machine learning. *The 26th Conference on Uncertainty in Artificial Intelligence*, pages 8–11.

[23] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA. ACM.

[24] Maruhashi, K., Guo, F., and Faloutsos, C. (2011). Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *Advances in Social Networks Analysis and Mining, 2011 International Conference on*, pages 203–210. IEEE.

[25] Newsome, J., Shi, E., Song, D., and Perrig, A. (2004). The sybil attack in sensor networks: analysis & defenses. In *Proceedings of the 3rd international symposium on Information processing in sensor networks*, pages 259–268. ACM.

[26] Noorden, R. V. (2013). Brazilian citation scheme outed. `http//facebook.com/note.php?note_id=76191543919`. october, 2014.

[27] Pandit, S., Chau, D. H., Wang, S., and Faloutsos, C. (2007). 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.

[28] Papalexakis, E. E., Beutel, A., and Steenkiste, P. (2012). Network anomaly detection using co-clustering. In *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining*, pages 403–410. IEEE Computer Society.

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

[30] Peeters, R. (2003). The maximum edge biclique problem is np-complete. *Discrete Applied Mathematics*, 131(3):651–654.

[31] Pei, J., Jiang, D., and Zhang, A. (2005). On mining cross-graph quasi-cliques. In *Proceedings of the eleventh ACM international conference on Knowledge discovery in data mining*, pages 228–238. ACM.

[32] Prakash, B. A., Sridharan, A., Seshadri, M., Machiraju, S., and Faloutsos, C. (2010). Eigenspokes: Surprising patterns and scalable community chipping in large graphs. In *Advances in Knowledge Discovery and Data Mining*, pages 435–448. Springer.

[33] Roy, A., Mihailovic, I., and Zwaenepoel, W. (2013). X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM.

[34] Su, X.-F., Zeng, H.-J., and Chen, Z. (2005). Finding group shilling in recommendation system. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 960–961. ACM.

[35] Tu, K. and Honavar, V. (2008). Unsupervised learning of probabilistic context-free grammar using iterative biclustering. In *International Colloquium on Grammatical Inference*, pages 224–237. Springer.

[36] Zarankiewicz, K. (1951). Problem p 101. In *Colloq. Math*, volume 2, page 5.