Fast Asynchronous Anti-TrustRank for Web Spam Detection

Joyce Jiyoung Whang∗
Sungkyunkwan University (SKKU)
Suwon, Korea
jjwhang@skku.edu

Yeon Seong Jeong
Sungkyunkwan University (SKKU)
Suwon, Korea
jys3548@skku.edu

Inderjit S. Dhillon
University of Texas at Austin
Austin, TX, USA
inderjit@cs.utexas.edu

Seonggoo Kang
Naver Corporation
Seongnam, Korea
seong-goo.kang@navercorp.com

Jungmin Lee
Naver Corporation
Seongnam, Korea
jmin.lee@navercorp.com

ABSTRACT

Web spam detection is an important problem in Web search. Since Web spam pages tend to have a lot of spurious links, many Web spam detection algorithms exploit the hyperlink structure between the Web pages to detect the spam pages. Anti-TrustRank algorithm is a well-known link-based spam detection algorithm which follows the principle that spam pages are likely to be referenced by other spam pages. Since a real-world Web graph involves tens of billions of nodes, it is crucial to develop work-efficient Web spam detection algorithms. In this paper, we develop asynchronous Anti-TrustRank algorithms which allow us to significantly reduce the number of arithmetic operations compared to the traditional synchronous Anti-TrustRank algorithm without degrading the performance in detecting Web spams. We theoretically prove the convergence of the asynchronous Anti-TrustRank algorithms, and conduct experiments on a real-world Web graph indexed by NAVER which is the most popular search engine in Korea.

ACM Reference Format:

1 INTRODUCTION

Web spam detection is one of the most important tasks in Web search. Given a Web graph with a set of nodes and edges where a node indicates a Web document and an edge indicates a hyperlink between documents, search engines rank the documents based on the link structure, e.g., [3], [9], and [11]. Web spams refer to the Web documents that have a lot of spurious links (e.g., creating link farms [21]) to mislead the search engines [7]. Therefore, it is critical for a search engine to correctly detect the Web spams to provide reliable search results.

∗Corresponding author.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/authors.

MIS2, 2018, Marina Del Rey, CA, USA
© 2018 Copyright held by the owner/authors.
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

A number of Web spam detection methods have been proposed over the years including link-based spam detection and content-based spam detection [17]. In particular, link analysis has been considered to be an important feature of a good spam detection method [1]. For example, [6] measures the impact of link spamming on a page’s rank, and [4] provides a way to detect nepotistic links.

TrustRank [7] is one of the well-known link-based spam detection methods. The main idea of the TrustRank method is that good pages tend to point to other good pages, which leads to computing TrustRank scores of the nodes. Similarly, the Anti-TrustRank algorithm [10] has also been proposed with an intuition that spam pages tend to be pointed by other spam pages. It has been shown that Anti-TrustRank is able to achieve a higher precision than the TrustRank method. A method of propagating both trust and distrust values also has been considered [22]. Indeed, computing the TrustRank or Anti-TrustRank scores can be interpreted as computing a personalized PageRank (8) (also called as a biased PageRank) on the Web graph with a set of carefully selected seeds. A number of variations of PageRank have been proposed to accelerate PageRank computations including [2], [13], [23], and [5]. In particular, [20] has proposed a multi-threaded data-driven PageRank algorithm.

Since a Web graph usually involves tens of billions of nodes, it is crucial to develop a work-efficient algorithm for Web spam detection. In this paper, we design asynchronous Anti-TrustRank algorithms which are able to significantly reduce the number of arithmetic operations compared to the traditional synchronous Anti-TrustRank method while achieving the same accuracy. We theoretically prove the convergence of the asynchronous Anti-TrustRank methods. On a real-world Web graph provided by Naver Corporation which is the largest search engine company in Korea, we empirically observe that our residual-based asynchronous Anti-TrustRank method while achieving the same accuracy. We theoretically prove the convergence of the asynchronous Anti-TrustRank methods. On a real-world Web graph provided by Naver Corporation which is the largest search engine company in Korea, we empirically observe that our residual-based asynchronous Anti-TrustRank method allows us to compute the Anti-TrustRank scores with only 10% of the arithmetic operations required for the synchronous Anti-TrustRank method without degrading the performance in detecting Web spams.

2 ANTI-TRUSTRANK ALGORITHM

The principle behind the Anti-TrustRank [10] algorithm is that spam pages are likely to be referred by other spam pages. Given a graph \( G = (V, E) \), the Anti-TrustRank method first selects a set of seeds which consists of manually examined spam documents. An Anti-TrustRank (ATR) score is assigned to each document such that a document with a high ATR score is considered as a spam...
Algorithm 1: Synchronous ATR
Input: $G' = (V', E'), S, α, ε
Output: ATR vector $x$
1: Initialize $x = (1 - α)e_s$
2: while true do
3: for $i ∈ V'$ do
4: if $i ∈ S$ then
5: $x^{new}_i = α \sum_{j ∈ Q_i} \frac{x_j}{|T_i|} + (1 - α) e_i$
6: else
7: $x^{new}_i = α \sum_{j ∈ Q_i} \frac{x_j}{|T_i|}$
8: end if
9: $\delta_i = |x^{new}_i - x_i|$
10: end for
11: $x = x^{new}$
12: if $\|\delta\|_∞ < ε$ then
13: break;
14: end if
15: end while
16: $x = \frac{x}{\|x\|_1}$

Algorithm 2: Asynchronous ATR
Input: $G' = (V', E'), S, α, ε
Output: ATR vector $x$
1: Initialize $x = (1 - α)e_s$
2: for $i ∈ V'$ do
3: $i ∈ S$ then
4: $x^{new}_i = α \sum_{j ∈ Q_i} \frac{x_j}{|T_i|} + (1 - α) e_i$
5: else
6: $x^{new}_i = α \sum_{j ∈ Q_i} \frac{x_j}{|T_i|}$
7: end if
8: end for
9: while (worklist is not empty)
10: $i = \text{worklist}.pop()$
11: if $i ∈ S$ then
12: $x^{new}_i = α \sum_{j ∈ Q_i} \frac{x_j}{|T_i|} + (1 - α) e_i$
13: else
14: $x^{new}_i = α \sum_{j ∈ Q_i} \frac{x_j}{|T_i|}$
15: end if
16: for $j ∈ T_i$ do
17: if $j$ is not in worklist then
18: worklist.push($j$)
19: end if
20: end while
21: $x = \frac{x}{\|x\|_1}$

Algorithm 3: Residual-based Asynchronous ATR
Input: $G' = (V', E'), S, α, ε
Output: ATR vector $x$
1: Initialize $x = (1 - α)e_s$
2: Initialize $r = (1 - α)αP^T e_s$
3: for $i ∈ V'$ do
4: $i ∈ S$ then
5: $r^{old}_i = r_i$
6: else
7: $r_i = r_i + \frac{r}{|T_i|}$
8: end if
9: for $j ∈ T_i$ do
10: if $r_j ≥ ε$ and $r^{old}_j < ε$ then
11: worklist.push($j$)
12: end if
13: end for
14: end for
15: $r_j = 0$
16: $r_j = 0$
17: end while
18: $x = \frac{x}{\|x\|_1}$

Document. The ATR scores of the seed spam documents are initialized to be one whereas the ATR scores of the rest of the nodes are initialized to be zero. From the seeds, the ATR scores are propagated to incoming neighbors of the nodes so that the documents having links to the spam documents end up with having high ATR scores.

2.1 Selecting Seeds
It is important to select good seeds in the Anti-TrustRank algorithm since the ATR scores are propagated from the selected seeds. One way to select good seed nodes is to consider PageRank scores [7], [10] because nodes with high PageRank scores are likely to be highly ranked by search engines, and it is critical to filter out spam documents which otherwise can be potentially exposed to users. Thus, we compute PageRank scores of the nodes, and select top-ranked nodes. Let $L$ denote the set of selected nodes. Then, human experts classify the nodes in $L$ into two classes: spam documents or normal documents. Let $S$ denote the set of spam documents among the nodes in $L$. Note that $S$ is a subset of $L$.

2.2 Synchronous Anti-TrustRank
The mechanism of how to compute the ATR scores is very similar to that of the personalized PageRank computation [8], [14]. The difference is that the ATR scores are propagated backward along with incoming links. Let $G' = (V', E')$ denote a graph with reverse edges, i.e., if an edge $(i,j) ∈ E$ then $(j,i) ∈ E'$. Also, let $A$ denote the adjacency matrix of $G'$. Then, computing ATR is identical to computing the personalized PageRank on $A$ with the personalized vector such that the positions of the seed spam documents (i.e., the nodes in $S$) have ones and other values are zeros. Let $Q_i$ denote the set of incoming neighbors of node $i$ on $G'$, and $T_i$ denote the set of outgoing neighbors of node $i$ on $G'$. Let $x$ denote a vector of the ATR scores, and $e_i$ denote a vector with ones for the positions of the seed spam documents and zeros for other positions. Also, let $α$ denote the damping factor (we use $α = 0.85$ throughout the paper), and $ε$ denote the tolerance. We assume that there is no self-loop in the graph, i.e., the diagonal elements of $A$ are all zeros. Algorithm 1 is a synchronous Anti-TrustRank algorithm where the ATR scores are updated only after all the nodes re-compute the ATR scores.

3 ASYNCHRONOUS ANTI-TRUSTRANK
We design asynchronous Anti-TrustRank algorithms by considering the Gauss-Seidel method. Instead of updating the ATR scores of all the nodes at every iteration, we maintain a worklist which contains a set of nodes whose ATR scores need to be updated. Initially, the worklist contains the entire vertices, and whenever we process a node from the worklist, we add the outgoing neighbors of the processed node (on $G'$) to the worklist. Indeed, for the global PageRank problem [3], a scalable data-driven PageRank algorithm [20] has been considered in a multi-threaded programming environment [16]. We extend this idea to the ATR computation which is shown in Algorithm 2.

3.1 Convergence of Asynchronous ATR
By extending the analysis of [15], we show the convergence of Algorithm 2.

Theorem 1. In Algorithm 2, when $x^{(k)}_i$ is updated to $x^{(k+1)}_i$, the total residual is decreased at least by $r_1(1 - α)$.

Proof. The ATR vector $x$ is computed as follows:

$$x = αP^T x + (1 - α)e_s$$

where $P$ is defined as $P = D^{-1}A$ ($D$ is the degree diagonal matrix) and $e_s$ is the personalized vector. This is the linear system of

$$(I - αP^T)x = (1 - α)e_s$$

and the residual is defined to be

$$r = (1 - α)e_s - (I - αP^T)x = αP^T x + (1 - α)e_s - x.$$

Let $x^{(k)}_i$ denote the $k$-th update of $x_i$. Since we initialize $x$ as $x = (1 - α)e_s$, the initial residual $r^{(0)}$ can be written as follows:

$$r^{(0)} = (1 - α)e_s - (I - αP^T)(1 - α)e_s = (1 - α)αP^T e_s ≥ 0. \quad (1)$$

For each node $i$ from the worklist, we update its ATR value as follows:
which is similar to the push-based data-driven PageRank in [20].

Now, by multiplying $e^T$ in (3), we get:

$$e^T r^{(k+1)} = \begin{cases} e^T r^{(k)} - r_i^{(k)} (1 - \alpha) & : T_i \neq \emptyset \\ e^T r^{(k)} - r_i^{(k)} (1 - \alpha) & : T_i = \emptyset \end{cases}$$

This implies that when a node $i$’s ATR value is updated, its residual $r_i$ becomes zero, and $ar_i/|T_i|$ is added to each of its outgoing neighbors’ residuals ($0 < \alpha < 1$). Thus, any step decreases the residual by at least $\gamma(1 - \alpha)$, and moves $x$ closer to the solution. □

**Theorem 2.** Algorithm 2 guarantees $\|r\|_{\infty} < \epsilon$ when it is converged.

**Proof.** Whenever a node’s ATR is updated, the residual of each of its outgoing neighbors is increased. Thus, if we ever change a node’s ATR, we need to add its outgoing neighbors to the worklist to verify that their residual is sufficiently small. This is what Algorithm 2 does. □

### 3.2 Residual-based Asynchronous ATR

Based on the analysis in Theorem 1, we note that the new ATR score of a node can be updated by just adding its current ATR score and its current residual by (2). To update the ATR scores in this way, we need to explicitly maintain the residual value for each node. Note that the residual of a node can be updated by (3). We design the residual-based asynchronous ATR algorithm shown in Algorithm 3 which is similar to the push-based data-driven PageRank in [20].

Let us compare Algorithm 3 and Algorithm 2. Whenever a node’s ATR score is updated, all of its outgoing neighbors (on $G'$) are pushed into the worklist in Algorithm 2. On the other hand, since we explicitly maintain the residual of each node in Algorithm 3, we can decide whether a node should be pushed to the worklist or not based on its residual. This can significantly reduce the unnecessary repeated computations.

### 4 EXPERIMENTAL RESULTS

We get a real-world Web graph from NAVER corporation which operates the most popular search engine called NAVER in Korea. We extract a subgraph of the entire Web graph using a variation of the forest fire graph sampling method [12]. Table 1 shows the basic statistics of the dataset. Among the 584,092 Web documents, 483,027 documents (82.7%) are labeled by human experts, i.e., those documents are manually classified into ‘spam’ or ‘normal’.

We first test the performance of the Anti-TrustRank algorithm in terms of detecting spam documents. Let $\text{sync}$ denote Algorithm 1, $\text{async}$ denote Algorithm 2, and $\text{rasync}$ denote Algorithm 3. As described in Section 2.1, human experts are supposed to manually label a subset of the nodes denoted by $\mathcal{L}$. Usually, the size of $\mathcal{L}$ is assumed to be very small since labeling requires human efforts. Among the nodes in $\mathcal{L}$, the set of spam documents $S$ is considered to be the seed nodes in the Anti-TrustRank algorithm. In our experiments, we assume that $p$ portion of the nodes can have labels among the entire vertex set $V$. That is, the number of labeled documents $|\mathcal{L}| = p|V|$. When we finish the Anti-TrustRank computation, we order the documents in descending order according to the Anti-TrustRank scores. A document with a high ATR score indicates that the document is likely to be a spam document. When we pick top $m$ documents, those documents are considered to be spam documents. Since the Anti-TrustRank algorithm computes a biased PageRank with the set $\mathcal{S}$, the Anti-TrustRank scores are propagated from the seed set $\mathcal{S}$. Thus, the number of documents associated with non-zero Anti-TrustRank scores is proportional to the size of $\mathcal{S}$.

We pick top $q|\mathcal{S}|$ documents (i.e., $m = q|\mathcal{S}|$) and count the number of spam documents, normal documents, and unlabeled documents among the retrieved documents. Table 2 shows the results with different $p$ and $q$ values. We set the tolerance parameter $\epsilon = 10^{-8}$, and notice that all the three methods, $\text{sync}$, $\text{async}$, and $\text{rasync}$, return the identical results for classifying the retrieved documents. We see that most of the retrieved documents are correctly classified into spam. This shows that in our dataset, the spam documents tend to be referred by other spam documents, which enables the Anti-TrustRank algorithm to work reasonably well.

Now, we investigate the computational cost of the $\text{sync}$, $\text{async}$, and $\text{rasync}$ methods. By varying $\epsilon$ and $p$, we count the number of ATR updates and the number of arithmetic operations required to make each method converge. Table 3 shows the results.

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>$i \in S$</td>
<td>$x_i^{(k+1)} = (1 - \alpha) + \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
<tr>
<td>Case 2</td>
<td>$i \notin S$</td>
<td>$x_i^{(k+1)} = \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AT R score</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updated, all of its outgoing neighbors (on $G'$) are pushed into the worklist in Algorithm 2. On the other hand, since we explicitly maintain the residual of each node in Algorithm 3, we can decide whether a node should be pushed to the worklist or not based on its residual. This can significantly reduce the unnecessary repeated computations.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync</td>
<td>Algorithm 1</td>
</tr>
<tr>
<td>async</td>
<td>Algorithm 2</td>
</tr>
<tr>
<td>rasync</td>
<td>Algorithm 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1: Real-world Web graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of normal documents</td>
</tr>
<tr>
<td>No. of spam documents</td>
</tr>
<tr>
<td>No. of unlabeled documents</td>
</tr>
<tr>
<td>No. of total documents</td>
</tr>
<tr>
<td>No. of links</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>$i \in S$</td>
<td>$x_i^{(k+1)} = (1 - \alpha) + \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
<tr>
<td>Case 2</td>
<td>$i \notin S$</td>
<td>$x_i^{(k+1)} = \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1: Real-world Web graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of normal documents</td>
</tr>
<tr>
<td>No. of spam documents</td>
</tr>
<tr>
<td>No. of unlabeled documents</td>
</tr>
<tr>
<td>No. of total documents</td>
</tr>
<tr>
<td>No. of links</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>$i \in S$</td>
<td>$x_i^{(k+1)} = (1 - \alpha) + \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
<tr>
<td>Case 2</td>
<td>$i \notin S$</td>
<td>$x_i^{(k+1)} = \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1: Real-world Web graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of normal documents</td>
</tr>
<tr>
<td>No. of spam documents</td>
</tr>
<tr>
<td>No. of unlabeled documents</td>
</tr>
<tr>
<td>No. of total documents</td>
</tr>
<tr>
<td>No. of links</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Condition</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>$i \in S$</td>
<td>$x_i^{(k+1)} = (1 - \alpha) + \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
<tr>
<td>Case 2</td>
<td>$i \notin S$</td>
<td>$x_i^{(k+1)} = \alpha \sum_{j \in Q} x_j^{(k)} /</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 1: Real-world Web graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of normal documents</td>
</tr>
<tr>
<td>No. of spam documents</td>
</tr>
<tr>
<td>No. of unlabeled documents</td>
</tr>
<tr>
<td>No. of total documents</td>
</tr>
<tr>
<td>No. of links</td>
</tr>
</tbody>
</table>
We first notice that the asynchronous algorithms, Async and Rasync, make much fewer ATR updates than the synchronous algorithm, Sync. This is because the asynchronous algorithms maintain a working set to selectively process the nodes whose ATR scores need to be updated while the synchronous algorithm processes all the nodes at every iteration. The number of ATR updates made in Async and Rasync should be the same because these methods follow the same rule to update the ATR scores.

In terms of the number of arithmetic operations, the asynchronous algorithms also save much computation compared to the synchronous algorithm. We compare Async and Rasync, we see that the residual-based asynchronous algorithm (Rasync) significantly reduces the number of arithmetic computations. As described in Section 3.2, Rasync is able to effectively reduce the size of the working set by exploiting the problem structure, which results in filtering out unnecessary computations.

### 5 CONCLUSIONS & FUTURE WORK

We develop asynchronous Anti-TrustRank algorithms which are shown to be effective in reducing the number of computations compared to the synchronous Anti-TrustRank algorithm on a real-world Web graph. While achieving the same precision with the synchronous Anti-TrustRank method for Web spam detection, the asynchronous Anti-TrustRank algorithms require much fewer Anti-TrustRank updates as well as arithmetic operations than the synchronous method. We plan to investigate the distinguishing characteristics of Web spams to incorporate them into the spam ranking system. Also, we intend to apply the idea of the asynchronous personalized PageRank computation to other graph mining applications such as community detection [19] and clustering [18].

### Acknowledgements

This research was supported by Naver Corp. and Basic Science Research Program through the NRF of Korea funded by MOE (2016R1A1B03394766 and NRF-2010-0020210) to JW. This work was also supported by US NSF grant IIS-1546452 to ID.

### REFERENCES


