HashNWalk: Hash and Random Walk Based Anomaly Detection in Hyperedge Streams (Supplementary Document)

Anonymous Author(s)

ABSTRACT

This document provides supplementary information to the main paper "HashNWalk: Sketch and Random Walk Based Anomaly Detection in Hyperedge Streams." We provide proofs and additional experimental deatils and results.

ACM Reference Format:

Anonymous Author(s). 2021. HashNWalk: Hash and Random Walk Based Anomaly Detection in Hyperedge Streams (Supplementary Document). In *Proceedings of (CIKM '21).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROOF OF LEMMA 2 & 3

In this section, we provide the proofs of Lemma 2 and Lemma 3 in the original paper. The transition probability from $\tilde{u}$ to $\tilde{v}$ is formalized as follow:

$$\tilde{P}_{\tilde{u}\tilde{v}}^{(m)} = \frac{\overbrace{\sum_{i=1}^{m} \alpha^{-t_i} \cdot \mathbb{1}(\tilde{u} \in \tilde{e}_i) \cdot \frac{\gamma_{\tilde{e}_i}(\tilde{v})}{\tilde{R}_{\tilde{e}_i}}}^{S_{\tilde{u}\tilde{v}}^{(m)}}}{\underbrace{\sum_{i=1}^{m} \alpha^{-t_i} \cdot \mathbb{1}(\tilde{u} \in \tilde{e}_i)}_{T_{\tilde{u}}^{(m)}}}$$

Proof of Lemma 2 By definition, $S_{uv}^{(m)}$ and $S_{uv}^{(m+1)}$ are

$$S_{uv}^{(m)} = \sum_{i=1}^{m} \alpha^{-t_i} \cdot \mathbb{1}(u \in \tilde{e}_i) \cdot \frac{\gamma_{\tilde{e}_i}(v)}{\tilde{R}_{\tilde{e}_i}}$$

and

$$S_{uv}^{(m+1)} = \sum_{i=1}^{m+1} \alpha^{-t_i} \cdot \mathbb{1}(u \in \tilde{e}_i) \cdot \frac{\gamma_{\tilde{e}_i}(v)}{\tilde{R}_{\tilde{e}_i}},$$

respectively. Thus,

$$S_{uv}^{(m+1)} - S_{uv}^{(m)} = \alpha^{-t_{m+1}} \cdot \mathbb{1}(u \in \tilde{e}_{m+1}) \cdot \frac{\gamma_{\tilde{e}_{m+1}}(v)}{\tilde{R}_{\tilde{e}_{m+1}}}.$$

Proof of Lemma 3 By definition, $T_u^{(m)}$ and $T_u^{(m+1)}$ are

$$T_u^{(m)} = \sum_{i=1}^{m} \alpha^{-t_i} \cdot \mathbb{1}(u \in \tilde{e}_i)$$

and

$$T_u^{(m+1)} = \sum_{i=1}^{m+1} \alpha^{-t_i} \cdot \mathbb{1}(u \in \tilde{e}_i),$$

respectively. Thus,

$$T_u^{(m+1)} - T_u^{(m)} = \alpha^{-t_{m+1}} \cdot \mathbb{1}(u \in \tilde{e}_{m+1}).$$

2 EXPERIMENTAL DETAILS

In this section, we discuss details of experimental settings.

Parameter Settings of InjectionU and InjectionB: The parameters we set for the injections are as follows:

- **email-Enron:** We set setup time to $t_{\text{setup}} = t_{100}$. In InjectionU, we set $g = 200$. In InjectionB, we set $k = 20$, $|N| = 5$, and $l = 10$.
- **tags-math:** We set setup time to $t_{\text{setup}} = t_{8000}$. In InjectionU, we set $g = 2000$. In InjectionB, we set $k = 20$, $|N| = 5$, and $l = 100$.
- **tags-overflow:** We set setup time to $t_{\text{setup}} = t_{100000}$. In InjectionU, we set $g = 100000$. In InjectionB, we set $k = 1000$, $|N| = 5$, and $l = 100$.
- **coauth-DBLP:** We set setup time to $t_{\text{setup}} = t_{30000}$. In InjectionU, we set $g = 100000$. In InjectionB, we set $k = 100$, $|N| = 5$, and $l = 1000$.

Note that the number of hyperedges to setup the model is less than 1% of the total hyperedges in all datasets.

Parameter Selection of the Methods: We discuss how we searched the hyperparameters of each method to obtain the best performance reported in the experimental results. For baselines, we included configurations requiring more space than ours. The range of configurations used for each method is as follows:

- **HashNWalk:** In `email-Enron`, we set $\alpha = 0.999$, $K = 15$, and $M = 20$. In `tags-math`, we set $\alpha = 0.999$, $K = 2$, and $M = 250$. In `tags-overflow`, we set $\alpha = 0.999$, $K = 2$, and $M = 2000$. In `coauth-DBLP`, we set $\alpha = 0.999$, $K = 1$, and $M = 2500$.
- **SedanSpot:** In `email-Enron`, we search from restart probability $\in \{0.10, 0.15, 0.20\}$, sample size $\in \{10000, 50000\}$, and number of random walks $\in \{50, 100\}$. In `tags-math`, we search from restart probability $\in \{0.10, 0.15, 0.20\}$, sample size $\in \{50000, 100000\}$, and number of random walks $\in \{50, 100\}$. In `tags-overflow`, we search from restart probability $\in \{0.10, 0.15, 0.20\}$, sample size $\in \{100000, 1000000\}$, and number of random walks $\in \{50, 100\}$. In `coauth-DBLP`, we search from restart probability $\in \{0.10, 0.15, 0.20\}$, number of random walks $\in \{50, 100\}$, and sample size $\in \{100000, 1000000\}$.

CIKM '21, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Precision and recall of detecting injected hyperedges in `coauth-DBLP` and `tags-overflow`. HashNWalk accurately detects both unexpected and bursty hyperedges. We do not report the results that takes more than an hour or is out of memory. Note that LSH uses 4 times of the space of the original hypergraphs.

	InjectionU								InjectionB							
	precision@				recall@				precision@				recall@			
Method	1000	2000	3000	4000	1000	2000	3000	4000	1000	2000	3000	4000	1000	2000	3000	4000
ideal	1.000	1.000	1.000	1.000	0.010	0.020	0.030	0.040	1.000	1.000	1.000	1.000	0.010	0.020	0.030	0.040
SedanSpot	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Midas	0.090	0.244	0.340	0.408	0.000	0.004	0.010	0.016	**1.000**	**1.000**	**1.000**	**1.000**	**0.010**	**0.020**	**0.030**	**0.040**
F-FADE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
LSH	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HashNWalk	**0.954**	**0.977**	**0.985**	**0.989**	**0.010**	**0.020**	**0.030**	**0.040**	**1.000**	**1.000**	**1.000**	**1.000**	**0.010**	**0.020**	**0.030**	**0.040**

(a) precision@k and recall@k in `tags-overflow`

	InjectionU								InjectionB							
	precision@				recall@				precision@				recall@			
Method	1000	2000	3000	4000	1000	2000	3000	4000	1000	2000	3000	4000	1000	2000	3000	4000
ideal	1.000	1.000	1.000	1.000	0.010	0.020	0.030	0.040	1.000	1.000	1.000	1.000	0.010	0.020	0.030	0.040
SedanSpot	0.064	0.054	0.049	0.047	0.000	0.001	0.001	0.001	0.019	0.018	0.019	0.017	0.000	0.000	0.000	0.000
Midas	0.284	0.303	0.317	0.322	0.002	0.006	0.009	0.012	**0.995**	**0.997**	**0.997**	**0.998**	**0.009**	**0.019**	**0.029**	**0.039**
F-FADE	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
LSH	**0.939**	**0.969**	0.875	0.656	**0.009**	**0.019**	0.026	0.026	0.378	0.189	0.126	0.094	0.003	0.003	0.003	0.003
HashNWalk	0.833	0.916	**0.944**	**0.958**	0.008	0.018	**0.028**	**0.038**	0.418	0.345	0.285	0.245	0.004	0.007	0.009	0.010

(b) precision@k and recall@k in `coauth-DBLP`

- **Midas:** In `email-Enron`, we search from number of buckets $\in \{5000, 10000, 20000\}$, number of hash functions $\in \{2, 8\}$, and decaying factor $\in \{0.3, 0.5, 0.7\}$. In `tags-math`, we search from number of buckets $\in \{10000, 50000, 100000\}$, number of hash functions $\in \{2, 8\}$, and decaying factor $\in \{0.3, 0.5, 0.7\}$. In `tags-overflow`, we search from number of buckets $\in \{100000, 1000000, 10000000\}$, number of hash functions $\in \{2, 8\}$, and decaying factor $\in \{0.3, 0.5, 0.7\}$. In `coauth-DBLP`, we search from number of buckets $\in \{100000, 1000000, 10000000\}$, number of hash functions $\in \{2, 8\}$, and decaying factor $\in \{0.3, 0.5, 0.7\}$.
- **F-FADE:** In `email-Enron`, we search from time interval for model update $\in \{100, 1000\}$, number of epochs $\in \{5, 10\}$, upper limit of memory size $\in \{10000, 20000\}$, online train steps $\in \{10, 20\}$, and cut-off threshold $\in \{0.1, 0.01, 0.001\}$. In `tags-math`, we search from time interval for model update $\in \{100, 1000\}$, number of epochs $\in \{5, 10\}$, upper limit of memory size $\in \{10000, 100000\}$, online train steps $\in \{10, 20\}$, and cut-off threshold $\in \{0.1, 0.01, 0.001\}$. In `tags-overflow`, we search from time interval for model update $\in \{100, 1000\}$, number of epochs $\in \{5, 10\}$, upper limit of memory size $\in \{1000000, 10000000, 100000000\}$, online train steps $\in \{10, 20\}$, and cut-off threshold $\in \{0.1, 0.01, 0.001\}$. In `coauth-DBLP`, we search from time interval for model update $\in \{100, 1000\}$, number of epochs $\in \{5, 10\}$, upper limit of memory size $\in \{1000000, 10000000, 100000000\}$, online train steps $\in \{10, 20\}$, and cut-off threshold $\in \{0.1, 0.01, 0.001\}$. The embedding size is fixed to 200 and the time decaying parameter is set to 0.999.
- **LSH:** For all datasets, we search from number of bands $\in \{2, 4, 8\}$ and the length of each band $\in \{2, 4, 8\}$.

Notably, HashNWalk covers 22.5%, 6.9%, 18%, and 60% of the space used in the original hypergraphs in `email-Enron`, `tags-math`, `tags-overflow`, and `coauth-DBLP`, respectively. Meanwhile, note that LSH uses 4 times of the space of the original hypergraphs.

3 ADDITIONAL EXPERIMENTS

Performance of HashNWalk: We provide experimental results on two datasets: `coauth-DBLP` and `tags-overflow`. In Table 1, we report precision and recall of detecting injected hyperedges in `coauth-DBLP` and `tags-overflow`. HashNWalk accurately detects both unexpected and bursty hyperedges injected by InjectionU and InjectionB, respectively. Furthermore, while some baselines took more than an hour or were killed due to out of memory, HashNWalk finished in several minutes using reasonable amount of space.

Case study on `cite-patent`: We provide the titles of the patents presented in Figure 4(b) in the main paper. The numbers correspond to those in the figure.

Unexpected patent

(1) Semiconductor integrated circuit and method for controlling semiconductor integrated circuit

Normal patents

(2) Pushing force deviating interface for damping mechanical vibrations
(3) Multistage compressor
(4) Hot swappable computer card carrier

Bursty patents

(5) Querying of copyright host, printing of copyright information and host registration of copyright data
(6) Sheet metal rocker arm, manufacturing method thereof, cam follower with said rocker arm, and assembling method thereof

(7) Turbine shroud thermal distortion control

<u>Case study on `tags-overflow`:</u> We share an experimental result on case study using `tags-overflow`. In the dataset, nodes are tags and hyperedges are the set of tags applied to a question. Some of the hyperedges whose score_{U} or score_{B} are high are listed as follow:

Set of *unexpected* keywords (high score_{U})

(1) channel / ignore / antlr / hidden / whitespace
(2) sifr / glyph / styling / text-styling / embedding
(3) retro-computing / boot / floppy / amiga
(4) robot / xmpp / sametime / ocs / aim
(5) onenote / ui-design / autosave / save / filesystems

Set of *bursty* keywords (high score_{B})

(1) python / javascript
(2) java / adobe / javascript
(3) c# / java
(4) jax-rs / java / javascript
(5) php / java

Notably, sets of unpopular tags tend to have high unexpectedness, while those that contain hot keywords, such as *Python* or *Javascript*, have high burstiness.