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

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 details and results.

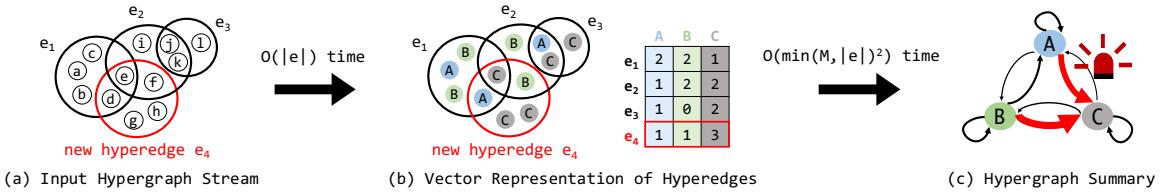


Figure 1: Outline of HashNWalk. (a) In a hyperedge stream, new hyperedges are constantly added. (b) Nodes are merged into supernodes with edge-dependent weights by hashing. (c) The hypergraph summary is updated in response to the new hyperedge, and based on the summary, the anomalousness of the hyperedge is measured.

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:

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

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

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

respectively. Thus,

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

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

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

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 settings of anomaly-detection algorithms: We discuss how we search the hyperparameters of each method to obtain the best performance reported in the experimental results. For baselines, we include configurations requiring more space than ours. The range of configurations used for each method is as follows:

- **HashNWalk:** 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 $\in \{10^5, 10^6\}$, 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 $\in \{10^5, 10^6\}$.
- **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 $\in \{10^6, 10^7, 10^8\}$, 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 $\in \{10^4, 10^5\}$, 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 $\in \{10^6, 10^7, 10^8\}$, 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 $\in \{10^6, 10^7, 10^8\}$, 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

Accuracy: We provide experimental results on three datasets: `tags-overflow`, `tags-math`, and `coauth-DBLP`. In Table 1, we report precision and recall of detecting injected hyperedges. HashNWalk accurately spots both unexpected and bursty hyperedges injected by InjectionU and InjectionB (see Section 5.2 of the main paper for InjectionU and InjectionB). respectively. Furthermore, while some baselines take more than an hour or were killed due to out of memory, HashNWalk terminate in several minutes using reasonable amount of space.

Table 1: Precision and recall of detecting injected hyperedges in `tags-overflow`, `tags-math`, and `coauth-DBLP`. 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	100	200	300	400	100	200	300	400	100	200	300	400	100	200	300	400
Ideal	1.000	1.000	1.000	1.000	0.050	0.100	0.150	0.200	1.000	1.000	1.000	1.000	0.050	0.100	0.150	0.200
SedanSpot	0.330	0.380	0.380	0.302	0.016	0.038	0.057	0.060	0.850	0.745	0.673	0.562	0.042	0.074	0.101	0.112
Midas	0.650	0.700	0.643	0.620	0.032	0.070	0.096	0.124	**1.000**	**1.000**	**1.000**	**1.000**	**0.050**	**0.100**	**0.150**	**0.200**
F-FADE	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	**1.000**	**1.000**	**1.000**	**1.000**	**0.050**	**0.100**	**0.150**	**0.200**
LSH	0.350	0.260	0.230	0.205	0.017	0.026	0.034	0.041	0.110	0.100	0.100	0.112	0.005	0.010	0.015	0.022
HashNWalk	**0.905**	**0.953**	**0.957**	**0.733**	**0.045**	**0.095**	**0.144**	**0.147**	0.999	0.999	0.998	0.996	**0.050**	**0.100**	**0.150**	0.199

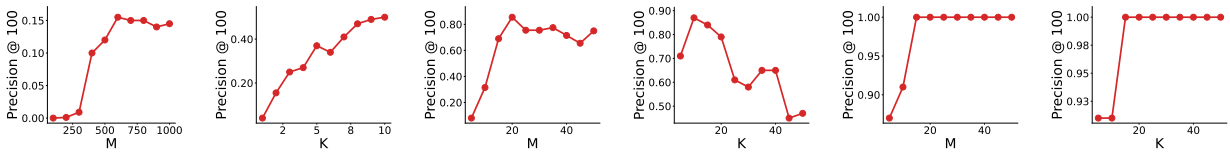
(b) precision@k and recall@k in `tags-math`

	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

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

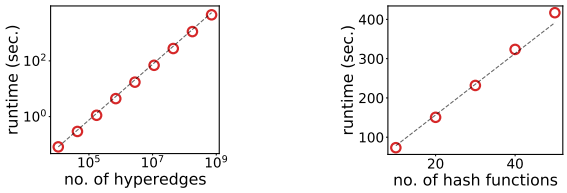
Parameter analysis: We evaluate HashNWalk with different numbers of supernodes M and hash functions K. As shown in Figure 2, the performance of HashNWalk depends on these parameters, and in most cases, there is a positive correlation with M and K.

Scalability: As shown in Figure 3, the total runtime of HashNWalk is linear in the number of hyperedges and the number of hash functions.



(a) M in TS (b) K in TS (c) M in SU (d) K in SU (e) M in SB (f) K in SB

Figure 2: The performance (specifically, Precision@100) of HashNWalk depends on the number of supernodes (M) and the number of hash functions (K). TS: `Transaction`. SU: `SyntheticU`. SB: `SyntheticB`.



(a) Number of hyperedges (b) Number of hash functions

Figure 3: The total runtime of HashNWalk is linear in the number of hyperedges ($\mathcal{E}$) and the number of hash functions (K) in the upscaled hypergraph of `email-Enron`.

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

Case study on `tags-overflow`: 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 $\mathsf{score}_\mathsf{U}$ or $\mathsf{score}_\mathsf{B}$ are high are listed as follow:

Set of *unexpected* keywords (high $\mathsf{score}_\mathsf{U}$)

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 $\mathsf{score}_\mathsf{B}$)

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.