Efficiently Answering k-hop Reachability Queries in Large Dynamic Graphs for Fraud Feature Extraction

Zequan Xu[1], Siqiang Luo[2], Jieming Shi[3], Hui Li[1], Chen Lin[1], Qihang Sun[4], and Shaofeng Hu[4]
[1]School of Informatics, Xiamen University, Xiamen, China; [2]Nanyang Technological University, Singapore
[3]The Hong Kong Polytechnic University, Hong Kong, China; [4]Tencent Inc., Guangzhou, China
xuzequan@stu.xmu.edu.cn, siqiang.luo@ntu.edu.sg, jieming.shi@polyu.edu.hk
{hui, chenlin}@xmu.edu.cn, {aaronqhsun, hugohu}@tencent.com

Abstract*—Instant messaging client (IMC) is now an essential tool for mobile users. In the representative IMC WeChat, cybercriminals deceive frauds, causing financial loss to normal users. Through statistical analysis, we find that certain fraud interactions commonly occur among WeChat users who are not k-hop neighbors. Therefore, efficiently answering whether the distance between two vertices is not longer than k at a certain time point (i.e., k-hop reachability queries) over the dynamic social graph of WeChat becomes a crucial task for fraud feature extraction in the detection system: it can help human experts quickly identify suspicious user interactions and the query results can be further used as the input feature to the downstream machine learning based detection methods. In this paper, we illustrate Bidirectional k-hop Reachability Query Processing over a Dynamic Graph (BREAD) that is used in WeChat for extracting the k-hop reachability feature for fraud detection. BREAD adopts the idea of estimating Personalized PageRank value. It first conducts the backward search from the destination vertex to construct an intermediate vertex set. Then, it performs a certain amount of random walks from the start vertex to see whether they can hit the intermediate vertex set, and the results are returned to answer k-hop reachability queries. We further propose BREAD++ that leverages the massive parallel processing power of GPU to achieve a considerable performance gain. Experiments on several large-scale dynamic graph benchmarks and the social graph of WeChat have demonstrated that BREAD/BREAD++ is superior than existing index-free competitors: our methods provide not only fast but also accurate responses and they are of practical value to k-hop reachability feature extraction in the fraud detection system of WeChat. Our implementation is available at https://github.com/XMUDM/BREAD.

***Index Terms*—k-hop reachability, personalized pagerank, fraud detection**

I. Introduction

Instant messaging client (IMC) has become indispensable to mobile users. WeChat[1], developed by Tencent, is a representative IMC with billions of active users. Attracted by the massive number of users, cybercriminals perpetrate frauds in WeChat for illegal profits, causing financial loss to normal users and affecting user experience.

Much effort has been devoted to detecting frauds. A useful feature to fraud detection is related to the *k-hop reachability query over a dynamic graph (KR query for short)*: given the directed social graph G with n vertices (i.e., users) and m edges (i.e., WeChat friend relations occur at certain time points), a KR query $\langle s, t, k, T\rangle$ asks whether there is a path from a source vertice s to a destination vertice t in G at time point T and the path is not longer than k hops. Through statistical analysis, we find that certain fraud interactions commonly occur among WeChat users who are *not* k-hop neighbors. Hence, the parameter k and the answer to KR queries can be used by human experts to judge whether some user interactions (e.g., chat and transactions) are suspicious. They can also be used as an input feature to the downstream machine learning based detection methods. Considering the large size of WeChat social graph and the importance of KR feature, efficiently answering KR queries becomes an vital task in the fraud detection system of WeChat.

There are few works studying efficiently answering KR queries [1], [2], [3], [4] and they rely on specially designed indexes that are not appropriate for the social graph of WeChat which is dynamically changing. On the other hand, a KR query is a special case of the *reachability query (R query)* $\langle s, t, T\rangle$ that asks whether a vertex s is reachable from another vertex t at time point T. There are two inefficient ways to answer such queries: (1) Traverse the entire graph using DFS or BFS. Such methods incur a high cost of $O(n+m)$ and are too slow for large graphs. (2) Precompute and materialize the transitive closure of the graph. Then, answer a R query by checking whether it exists in the transitive closure. Although the query can be answered in $O(1)$, the storage consumption of the transitive closure is $O(n^2)$ and it is prohibitive for large graphs. In the literature, some approaches try to balance query time and storage cost [5]. These methods construct and leverage an index that requires less space than the naive transitive closure-based method, and they have query processing time in the range of $O(1)$ and $O(n+m)$, where the former is the query time using the transitive closure and the later is the query time using DFS or BFS.

In this paper, we illustrate Bidirectional k-hop Reachability Query Processing over a Dynamic Graph (BREAD) that is used to extract KR feature in WeChat for fraud detection. BREAD adopts the idea of estimating Personalized PageRank (PPR) value [6]. The PPR value $\pi(u, v)$ of a vertex u with respect to a vertex v is the probability that a random walk starting from u terminates at v and it is a proximity measure used in various graph-based applications [7]. The contributions

Work done when the first author was an intern at Tencent. Hui Li is the corresponding author.

[1]https://www.wechat.com/en

of this paper are summarized as follows:

1) We propose BREAD for efficiently answering KR queries over large dynamic graphs. BREAD firstly adopt backward search from the destination vertex to extract intermediate vertex set. Then it performs random walks from the source to see whether they can hit intermediate vertices, and the result are returned as the answer.
2) The time complexity of BREAD for large-scale graphs is approximately $O(\sqrt{m \log n})$. It is smaller than the complexity of other competitors on large graphs.
3) We further propose BREAD++ that leverages the massive parallel processing power of GPU to achieve a considerable performance gain.
4) We have conducted extensive experiments on several large-scale dynamic graph benchmarks including the social graph of WeChat. Results show that our methods are superior than existing index-free competitors, and they can be used to efficiently assist fraud detection in WeChat.

II. Preliminaries

A. Problem Definition

A dynamic directed graph $G = (V, E)$ with $|V| = n$ and $|E| = m$ is a sequences of graph snapshots, or in other words, an initial graph along with a graph stream. If the graph is weighted, each edge $e = \{u \to v\}$ has a positive weight $w(u, v)$; otherwise we define $w(u, v) = \frac{1}{N_{in}(v)}$ where $N_{in}(v)$ is the neighbor set of v. Each edge/vertex is associated with a set of time points indicating when they are valid. A KR query $\langle s, t, k, T\rangle$ takes a dynamic graph G, a source s, a target t and a timestamp T as input, and asks whether there exists a path from s to t at T that is not longer than k. A R query $\langle s, t, T\rangle$ is the general case of the KR query without the constraint on path length.

B. Index-free Reachability Query Processing

In an evolving graph such as the social graph of IMC, constructing indexes may not be an appropriate solution since data changes require rebuilding some indexes. Consequently, we mainly consider index-free methods:

(1) ARROW [8]. ARROW is the state-of-the-art index-free method for answering R queries. At query time, ARROW conducts multiple random walks of fixed length from both source s and target t to construct two sets of 'stops': $F(s) = \{u : s \to u\}$ of vertices that are reachable from s as well as $B(t) = \{v : v \to t\}$ of nodes that can reach t. If there exists a vertex $w \in F(s) \cap B(t)$ (i.e., s can reach t with a path passing through w), the algorithm reports true, otherwise false. The most crucial part in ARROW is how to set the number of random walks and the fixed length of the random walks to balance query efficiency and accuracy. With a thorough theoretical analysis, ARROW chooses parameters walk length $l = c_{\text{walkLength}} \times diam$ and walk times $r = c_{\text{numWalks}} \times \sqrt[3]{n^2 \ln n}$, where $c_{\text{walkLength}}$ and c_{numWalks} are both parameters, and $diam$ is the diameter of the graph. The size of sets $F(s)$ and $B(t)$, and the time spent on their construction and intersection are all rl. The time complexity of ARROW for answering R queries is $O(\sqrt[3]{n^2 \ln n} \cdot diam)$ [8]. To extend ARROW to answering KR queries, we skip the preprocessing step of estimating diameter in ARROW and set walk length l as k. It suggests that k implies a possible distance between s and t if they are connected and any path with length greater than k can not be the answer.

After we deploy ARROW in our fraud detection system, we find one shortcoming of ARROW: the walk length depends on the diameter $diam$ of the input graph, which is the maximum distance between any pair of vertices. ARROW estimates the diameter for the initial graph and uses it as the upper bound on the diameter of the graph for all subsequent snapshots. Unfortunately, there are no known, efficient algorithms that can precisely estimate $diam$ [9].

(2) RWBFS. We modify ARROW and propose an alternative by constructing stop sets using both BFS and random walks. We call it RWBFS ('RW' is short for random walk). More specifically, we conduct random walks from the source vertex s to retrieve the stop set $F(s)$, and conduct BFS from the target t to form the stop set $B(t)$. This way, RWBFS increases the probability of $F(s) \cap B(t) \neq \varnothing$.

Conducting BFS without early stopping is rather inefficient. Therefore, we further adopt an early-stopping criteria in RWBFS to achieve a balance between accuracy and efficiency. Intuitively, conducting BFS for a vertex with high degree is time-consuming since it will push much more vertices to the queue for future traverse. To alleviate this problem, during the BFS process, if the degree of current visiting vertex is greater than a given threshold, we stop subsequent BFS that starts from this vertex. We choose $c \cdot \bar{d}$ as the threshold, where c is a parameter to be set and $\bar{d}$ is the average degree of vertices in the graph. However, this design does not have any theoretical guarantee for the preciseness of the query result and c still needs to be tuned.

III. Our Proposed Method BREAD

We first give an overview of BREAD in Sec. III-A. Then, as answering KR queries can be viewed as the extension to the classic reachability problem with a constraint on path, we first illustrate the basic version of BREAD for answering R queries in Sec. III-B. After that, we will illustrate how the basic version can be extended to answer KR queries in Sec. III-C. Finally, we analyze parameter selection and complexity of BREAD in Sec. III-D.

A. Overview

At a high level, at query time, BREAD first conducts backward search from the destination vertex t and then it performs random walks forwards from the source vertex s (i.e., forward search). The backward search starting from t will find a suitably large set of intermediate vertices that are near t. The purpose of forward search is to generate sufficient amount of random walks from s to see whether they can hit node(s) in the intermediate vertex set.

Algorithm 1: BackwardSearch(t, r_{max}, α)

```
Input: Graph $G$ with edge weights $w(u,v)$, target $t$, probability $\alpha$,
       residue threshold $r_{max}$
Output: $r(t,v), p(t,v)$ for all $v \in V$
1  $r(t,t) \leftarrow 1, r(t,v) \leftarrow 0$ for all $v \neq t$;
2  $p(t,v) \leftarrow 0$ for all $v$;
3  Candidate set $S \leftarrow \{t\}$;
4  while $S \neq \emptyset$ do
5      Temporary set $S' \leftarrow \emptyset$;
6      for $v \in S$ do
7          for $u \in N_{in}(v)$ do
8              $r(t,u) \leftarrow r(t,u) + (1-\alpha) \cdot w(u,v) \cdot r(t,v)$;
9              if $r(t,u) > r_{max}$ then
10                 $S' \leftarrow S' \cup \{u\}$;
11         $p(t,v) \leftarrow p(t,v) + \alpha \cdot r(t,v)$;
12         $r(t,v) \leftarrow 0$;
13     $S \leftarrow S'$;
```

Algorithm 2: Answer R Queries

```
Input: Graph $G$, teleport probability $\alpha$, source $s$, target $t$, residue
       threshold $r_{max}$
Output: Answer to the R query $\langle s,t,T \rangle$
1  $r(t,v), p(t,v)$ = BackwardSearch($t, r_{max}, \alpha$);
2  if $p(t,s) > 0$ then
3      answer $\leftarrow$ true;
4  else
5      number of walks $\omega \leftarrow c \cdot r_{max}/\delta$, answer $\leftarrow$ false;
6      for index $i \in [\omega]$ do
7          $cur \leftarrow s$;
8          while true do
9              if rand() $< \alpha$ then
10                 break;
11             Sample $v$ from $cur$'s out-neighbor;
12             if $r(t,v) > 0$ then
13                 answer $\leftarrow$ true;
14                 break;
15             $cur \leftarrow v$;
```

B. Answering R Queries

The reachability problem has connection with the definition of PPR. The PPR value $\pi(s,t)$ of vertex s with respect to t can be viewed as the probability that a random walk starts from s and terminates at t. From another perspective, if $\pi(s,t)$ is larger than zero, there exist random walks that can reach t, i.e., s and t are connected. Moreover, ARROW [8] illustrated in Sec. II-B actually resembles the Monte Carlo method to estimate $\pi(s,t)$. Thus, we are inspired to answer the R query $\langle s,t \rangle$ by approximating the PPR value $\pi(s,t)$ and we adapt the idea of the bidirectional PPR estimator [6] to this end.

The first step of BREAD for answering R queries is to traverse backwards from the destination vertex t to construct a set of intermediate vertices that are close to t. Alg. 1 illustrates this process. This step is based on the Approx-Contributions algorithm [10]. Given a target vertex t, we first initialize its residual value $r(t,t)$ as 1, then perform a while loop to spread this value to its neighboring vertices (lines 4-13). For any vertex v with non-zero residual value, we transfer its residual value to its neighboring vertices $u \in N_{in}(v)$ by an attenuation factor $(1-\alpha) \cdot w(u,v)$ (line 8), and keep $\alpha \cdot r(t,v)$ as its own reserve value $p(t,v)$ (line 11). The loop terminates when there is no vertex of which the residual value is greater than the predefined threshold r_{max}. The crucial part for this step is how to choose a proper value for the residue threshold r_{max} in order to cover a suitable amount of vertices while maintaining efficiency. We set $r_{max} = \epsilon\sqrt{\frac{\bar{d}\delta}{\ln(2/p_{\text{fail}})}}$, where p_{fail} is the failure probability, $\bar{d}$ is the average degree of vertices in the graph, ϵ is the relative error, and δ is the minimum PPR we want to accurately estimate in the bidirectional PPR estimator. δ can be viewed as a parameter to control the accuracy.

Then, BREAD leverages residual values $r(t,v)$ and reserve values $p(t,v)$ for all vertices $v \neq t$ from the first step to answer reachability queries. Alg. 2 depicts this step. BREAD first checks whether the reserve value $p(t,s)$ of the source vertex is greater than zero (line 2). If $p(t,s)$ is non-zero, it indicates that the path(s) connecting s and t have already been discovered in the backward search process. Otherwise, we conduct random walks from the source s (forward search). Once a random walk visits a vertex v with non-zero residual value, BREAD returns true, indicating the source vertex s can be connected to target t via vertex v. The core part of this process is how to set a proper value for the number of random walks ω to maintain a balance between accuracy and efficiency. We set $\omega = c \cdot r_{max}/\delta$, where c is a parameter controlling the accuracy. Note that BREAD performs random walks under the specific temporal constraint T of the query, i.e., edges along the path that a random walk traverse should exist at the time point T. Compared to ARROW, BREAD does not require the exact value or the estimation of the diameter when performing random walks, and thus it avoids the time-consuming preprocessing step in ARROW.

C. Answering KR Queries

The difference between KR queries and R queries is the constraint on the length of path(s) connecting the source s and the target t. Answering KR queries is more challenging since there may exist multiple viable paths between s and t but only part of them satisfy the constraint that the path length is not longer than k.

To answer KR queries, we modify the basic version of BREAD in Sec. III-B. As illustrated in Alg. 3, in order to examine whether a path found by BREAD satisfies the hop constraint, BREAD maintains an extra variable $d(t,v)$ for each vertex v in the backward search process. It records the shortest distance seen so far between v and t (line 10). When BREAD conducts random walks forwards (Alg. 4), once a random walk with length l visits a vertex v with a non-zero residual value, BREAD additionally checks whether the sum of length l and the shortest distance from v to t is greater than the given constraint k (line 12). If so, the algorithm returns true, otherwise false. Although the hop constraint in KR queries brings additional challenges, we can utilize the constraint to guide the random walk to improve efficiency. Once the length of a random walk exceeds k, BREAD terminates the process as it is impossible for the current random walk to find a path that conforms to the hop constraint (lines 17-18).

Algorithm 3: KBackwardSearch(t, r_{max}, α)

```
Input: Graph $G$ with edge weights $w(u,v)$, target $t$, probability $\alpha$,
       residue threshold $r_{max}$
Output: $r(t,v), p(t,v), d(t,v)$ for all $v \in V$
1  $r(t,t) \leftarrow 1, r(t,v) \leftarrow 0$ for all $v \neq t$;
2  $p(t,v) \leftarrow 0$ for all $v$;
3  Candidate set $S \leftarrow \{t\}$;
4  $d(t,t) \leftarrow 0, d(t,v) \leftarrow \inf$ for all $v \neq t$;
5  while $S \neq \emptyset$ do
6      Temporary set $S' \leftarrow \emptyset$;
7      for $v \in S$ do
8          for $u \in N_{in}(v)$ do
9              $r(t,u) \leftarrow r(t,u) + (1-\alpha)w(u,v)r(t,v)$;
10             $d(t,u) \leftarrow \min\big(d(t,u), d(t,v)+1\big)$;
11             if $r(t,u) > r_{max}$ then
12                 $S' \leftarrow S' \cup \{u\}$;
13         $p(t,v) \leftarrow p(t,v) + \alpha r(t,v)$;
14         $r(t,v) \leftarrow 0$;
15     $S \leftarrow S'$;
```

The hop constraint k is also beneficial to guiding the random walks in ARROW. When answering KR queries using ARROW, we omit the preprocessing step for estimating the graph diameter and directly use k as the walk length l. Since k may be larger than the diameter estimated by ARROW, it may increase the probability that, when answering the KR query $\langle s,t,k,T \rangle$, the two stop sets $F(s)$ and $B(t)$ have an intersection. Similar strategy can be used in RWBFS.

D. Analysis of Parameters and Complexity

In Alg. 2, once a random walk from s finds a vertex v marked by t in the previous backward search stage, the algorithm will terminate and return true as the answer to the current reachability query. As mentioned above, answering R queries resembles the calculation of PPR values. BREAD maintains an extra backward reserve vector $p(t)$ for t in backward search process. Every time BREAD sees a vertex v satisfying the push condition, BREAD increases v's backward reserve by $\alpha \cdot r(t,v)$. During the Monte Carlo process, BREAD terminates once we found a vertex with residual value greater than zero. If we change it to record the residual value of the final node of the i-th random walk X_i as the bidirectional PPR estimator [6], then the following theorem is satisfied (Detailed analysis can be found in [6]):

Theorem 1: Given a start node s (or source distribution σ), a target node t, minimum PPR δ, maximum residual $r_{max} > \frac{2e\delta}{\alpha\epsilon}$, relative error $\epsilon \leq 1$, and failure probability p_{fail}, the bidirectional PPR estimator ouputs an estimation $\hat{\pi}(s,t)$ with probability at least $1 - p_{\text{fail}}$ the following hold:

- If $\pi(s,t) \geq \delta$: $|\pi(s,t) - \hat{\pi}(s,t)| \leq \epsilon\pi(s,t)$
- If $\pi(s,t) \leq \delta$: $|\pi(s,t) - \hat{\pi}(s,t)| \leq 2e\delta$

where $\hat{\pi}(s,t) = p(t,s) + \frac{1}{\omega}\sum_{i=1}^{\omega} X_i$.

With $c = \frac{3}{\epsilon^2}\ln\frac{2}{p_{\text{fail}}}$ and $\omega = c\frac{r_{max}}{\delta}$ that are later used in our experiments, Theorem 1 ensures that $|\pi(v,t) - \hat{\pi}(v,t)| < r_{max}$ for any v in the graph. Therefore, for a reachable vertex s from t, $\pi(s,t)$ is greater than zero and the approximation error for $\hat{\pi}(v,t)$ is less than r_{max}, which is a strong guarantee for answering the k-hop reachability query accurately.

Algorithm 4: Answer KR Queries

```
Input: Graph $G$, teleport probability $\alpha$, source $s$, target $t$, hop $k$,
       residue threshold $r_{max}$
Output: Answer to the $k$-hop reachability query $\langle s,t,k \rangle$
1  $r(t,v), p(t,v), d(t,v)$ = KBackwardSearch($t, r_{max}, \alpha$);
2  if $p(t,s) > 0$ and $d(t,s) \leq k$ then
3      answer$\leftarrow$ true;
4  else
5      number of walks $\omega \leftarrow c \cdot r_{max}/\delta$, answer $\leftarrow$ false, $l \leftarrow 0$;
6      for index $i \in [\omega]$ do
7          $cur \leftarrow s$;
8          while true do
9              if rand() $< \alpha$ then
10                 break;
11             Sample $v$ from $cur$'s out-neighbor;
12             if $r(t,v) > 0$ and $l + d(t,v) \leq k$ then
13                 answer $\leftarrow$ true;
14                 break;
15             $cur \leftarrow v$;
16             $l \leftarrow l + 1$;
17             if $l > k$ then
18                 break;
```

Since BREAD adopts the idea of the bidirectional PPR estimator [6] and it further maintains an extra variable $d(t,v)$ for each vertex v in the backward search process for later use in the early stopping of forward search, the complexity of BREAD is not larger than the complexity of estimating PPR value using the bidirectional PPR estimator, i.e., $O(\sqrt{\frac{\bar{d}}{\delta}}\frac{\sqrt{\log(1/p_{\text{fail}})}}{\alpha\epsilon})$. Since δ and p_{fail} are usually set as $1/n$ [6], the time complexity of BREAD is approximately $O(\sqrt{m\log n})$, which is less than the time complexity $O(\sqrt[3]{n^2\ln n} \cdot diam)$ of ARROW [8] in a large-scale graph.

IV. Enhanced BREAD with GPU

We further propose BREAD++ that uses the massive parallel processing power of the GPU to further boost BREAD.

Sparse representation. Backward search on the CPU can be implemented utilizing data structures such as heaps, since they can be efficiently updated. However, these complex data structures are not available on the GPU. The prevalent data structure used on the GPU is array. In order to efficiently leverage GPU memory, we adopt the Compressed Sparse Row (CSR) format to store graph data.

Parallel Backward Search. The key of the backward search algorithm is to maintain a candidate set S with vertices satisfying the push condition and update the corresponding values based on Alg. 1. While on the CPU we can use data structure like priority queue to efficiently maintain S and terminate the procedure when there are no vertices that satisfy the condition, we need a new solution based on array for GPU.

When implementing backward search on the GPU, we divide the whole procedure into three steps which are illustrated in Alg. 5. Prior to the residual push procedure, we first allocate the memory for the residue vector and the reserve vector on the GPU and then use the same initialization as on the CPU.

Algorithm 5: Parallel BREAD

```
Input: Graph $G$ with edge weights $w(u, v)$, source $s$, target $t$,
       teleport probability $\alpha$, residue threshold $r_{max}$
Output: Dynamic reachability answer for query $s \to t$
1  $r(t, t) \leftarrow 1, r(t, v) \leftarrow 0$ for all $v \neq t$;
2  $p(t, v) \leftarrow 0$ for all $v$;
3  candidate set $S \leftarrow \{t\}$;
4  while $S \neq \emptyset$ do
5      $F$ = UpdateNextFrontierSetFlags($r(t, v), r_{max}$);
6      $S$ = GetNextFrontierSet($F$);
7      for $v \in S$ do
8          $p(t, v) \leftarrow p(t, v) + \alpha r(t, v)$;
9          for $u \in N_{in}(v)$ do
10             atomicAdd($r(t, u), (1 - \alpha)w(u, v)r(t, v)$)
11         $r(t, v) \leftarrow 0$;
12 if $p(t, s) > 0$ then
13     answer $\leftarrow$ true;
14 else
15     number of walks $\omega \leftarrow c \cdot r_{max}/\delta$, answer $\leftarrow$ false;
16     parallel for index $i \in [\omega]$ do
17         $cur \leftarrow s$;
18         while true do
19             if rand() $< \alpha$ then
20                 break;
21             Sample $v$ from $cur$'s out-neighbor;
22             if $r(t, v) > 0$ then
23                 answer $\leftarrow$ true;
24                 break;
25             $cur \leftarrow v$;
```

The first step in each iteration is to obtain the candidate set S prepared for the next step residual push. There are multiple ways to implement it in parallel: (1) One possible method is to use an array F with size n where each element represents a candidate flag for each vertex v in the graph. Once the residual value $r(t, v)$ becomes larger than the residual threshold r_{max} during backward search, we set its corresponding flag to 1. We then adopt a standard parallel select to collect all the flagged vertices in F and add them to the candidate set S. (2) Another way to achieve this is using the assistance of atomic operations. We can dynamically detect and insert candidate vertex into set S during backward search. In order to do so, an index is used to point to the next available position for insertion. If a vertex v satisfies the push condition, the thread handling v will atomically place it at the index position and increase the index by 1. The second implementation involves atomic operations, which may affect the efficiency if there exist many vertices to push. Considering that real-world graphs are mostly in a large scale, we decide to retrieve the candidate set in parallel using the first method.

The last step is to update the residual value of each candidate's neighboring vertices as well as its own reserve value. Here, we utilize atomic operations to ensure that the residual values are correctly transferred. We do not require atomic operations when updating candidate's reserve value as it will not cause any conflict that incurs an error. An alternative way to avoid using atomic operations is to make use of the sort and reduce algorithm. Given key-value pairs consisting of neighbor vertex ids and their corresponding residual values, the sort and reduce algorithm first runs a parallel sort on pairs. Then, it conducts a parallel reduce to aggregate the residual values with the same key and transfer results to corresponding vertices. However, this method will lead to significant overhead when encountering large frontiers [11]. Therefore, we take advantage of atomic operations in our implementation.

TABLE I
STATISTICS OF DATASETS.

Name	$\|V\|$	$\|E\|$	d_{max}	D	D_{90}
Digg	279.6K	1.7M	12.2K	18	4.96
BibSonomy	210.4K	2.5M	428.4K	12	4.01
Flickr	2.3M	33.1M	34.1K	23	6.88
Stackoverflow	2.6M	63.4M	194.8K	11	4.41
Delicious	5.3M	301.1M	4.3M	14	4.58
Wiki	6.9M	129.8M	1.8M	16	4.47
Google	28.9M	462.9M	452K	22	6.16
WeChat	15.9M	627.8M	11.9K	30	7.94

Parallel Monte Carlo Process (Forward Search). After obtaining the intermediate vertices near the destination vertex t, we perform random walks from the start vertex s in parallel to examine whether any random walks can hit the intermediate vertices found in backward search. We omit the details here as the procedure is similar to conducting random walks on the CPU. To reduce the time spent on transferring data between CPU and GPU, we initialize the data and transfer them to the GPU before processing the query. After processing, query response is transferred back to the CPU.

Based on the above three solutions to answering R queries using GPU, we make some modifications to handle KR queries and propose BREAD++. During the procedure of parallel backward search, we maintain an extra array d of size n where each element $d(t, v)$ records the shortest distance for each vertex v from target t. When a random walk with length l visits a vertex v with non-zero residual value, BREAD++ will return true only if the sum of l and $d(t, v)$ is not greater than the given hop k. We are aware that some advanced techniques estimate PPR values in parallel on the CPU or the GPU [12], [11]. They are orthogonal to our work for KR queries and we may use them to further improve our methods. We leave these improvements as the future work.

V. EXPERIMENTS

A. Experimental Settings

Data. We use eight real-world large dynamic graphs. Tab. I shows statistics of the data. D_{90} means the 90-percentile effective diameter and D denotes the graph diameter. Datasets except WeChat are from the Konect repository [13]. WeChat dataset contains part of the social graph of WeChat users from Fujian province in China and we use it to evaluate whether BREAD/BREAD++ can provide effective support to fraud detection in WeChat. Test queries are generated using BFS. For a vertex s, performing BFS can obtain the subgraph G_s of s. The shortest distance $d(s, t)$ for every vertex t inside G_s is also stored during BFS. These information are then utilized for query generation in different task.

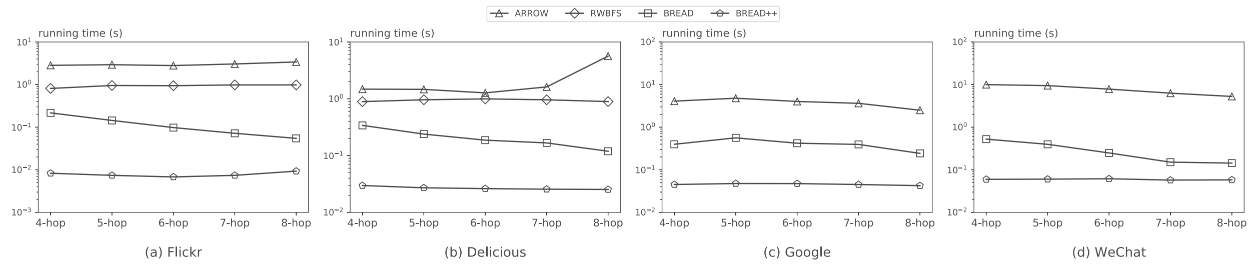
Fig. 1. Running time for answering k-hop reachability queries: varying k.

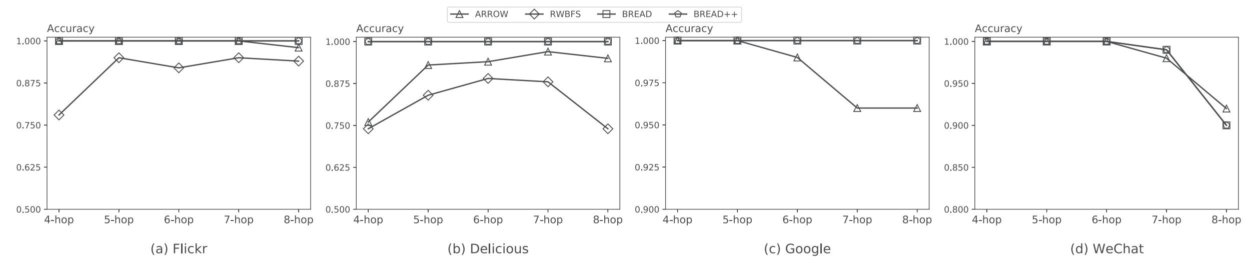
Fig. 2. Accuracy for answering k-hop reachability queries: varying k.

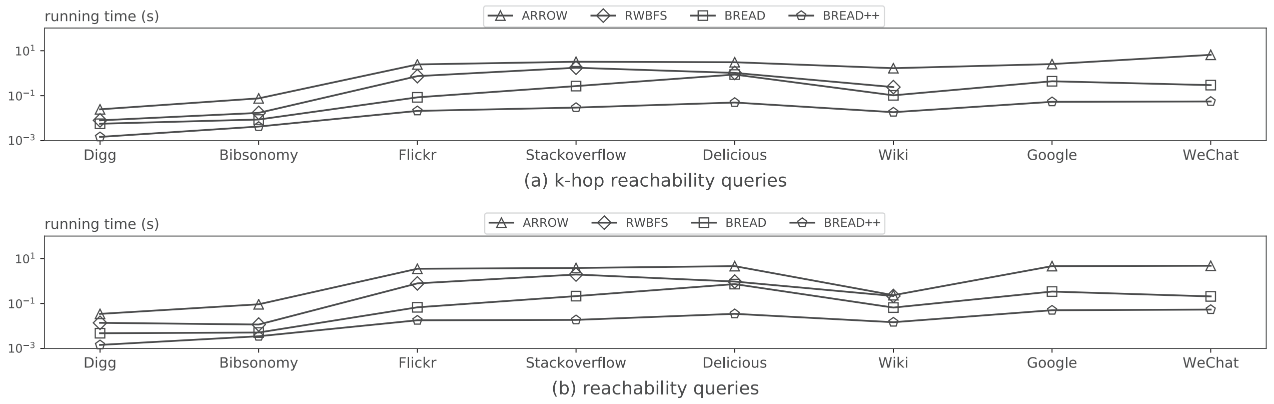
Fig. 3. Running time for the challenging task on different datasets.

Metrics. We use average query time and accuracy for evaluation. Note that, *if the total query time of a method on a dataset exceeds 2 hours, its results will not be reported.*

Methods. We compare BREAD/BREAD++[2] with ARROW[3] and RWBFS illustrated in Sec. II-B. For ARROW, following the original paper, we set $c_{walkLength}$ to 1 and select $c_{numWalks}$ among $[0.01, 0.05, 0.1, 0.5]$ to achieve good performance. For RWBFS, we use the same parameter setting as ARROW. For BREAD, we find that multiplying r_{max} by a factor $c_{r_{max}} = 1/16$ achieves a good balance between efficiency and accuracy.

Environment. We implement BREAD in C++ and compile it using GCC 7.5 with -O3 flag. BREAD++ is implemented and compiled using Nvidia CUDA 10.1 with -O3 flag. All experiments are performed on a Linux machine with 48 threads powered by one 12-core Intel Xeon(R) E5-2678 v3@2.50GHz, 256GB memory, and a Nvidia RTX2080Ti-12GB GPU.

[2]Implementation is available at https://github.com/XMUDM/BREAD.
[3]https://github.com/senguptaneha/temporalReachabilityC

B. Answering KR queries

Normal KR queries. We first show the effectiveness of BREAD/BREAD++ in answering normal KR queries. As the 90-percentile effective diameter D_{90} for all eight datasets lies approximately within $[4, 8]$, we choose k from $[4, 5, 6, 7, 8]$ and generate 100 queries for each k on each dataset. Fig. 1 and Fig. 2 report the average running time and accuracy of each method for answering 100 queries on four datasets: Flickr, Delicious, Google and WeChat. Note that the y-axis is in log-scale. We can see that RWBFS sometimes achieves similar accuracy as ARROW and it requires less time. However, the naive early-stopping criteria used in RWBFS does not perform well on large graphs like Google and WeChat: the running time exceeds 2 hours and it is not shown in Fig. 1. This is probably because that RWBFS treats vertices with larger degree equally, regardless of their distance from the target vertex t. BREAD overcomes this issue through spreading the residual value of t by an attenuation factor. Observe that BREAD achieves better performance than ARROW on all four datasets, which is consistent with our analysis on the time complexity of these

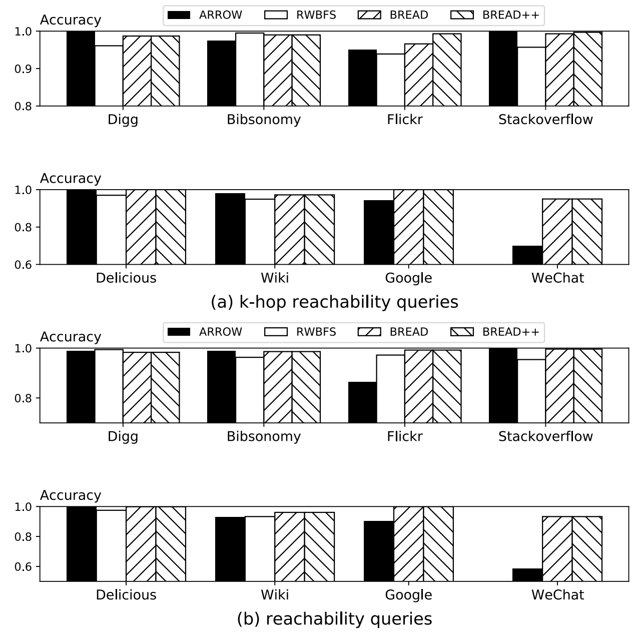

Fig. 4. Accuracy for the challenging task.

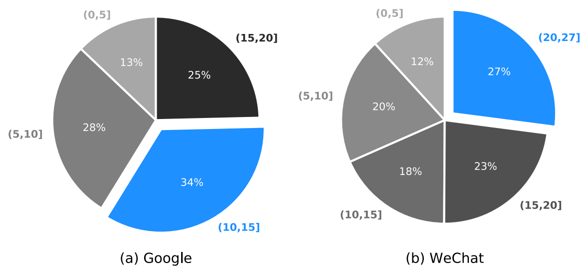

Fig. 5. Hop statistics for positive queries.

two algorithms in Sec. III-D. BREAD can achieve comparable or better accuracy than ARROW while it is an order of magnitude faster than ARROW. Finally, the GPU version of BREAD, BREAD++, achieves speedup over BREAD by using the massive computation power of the GPU. It is worthy pointing out that, in WeChat data, BREAD/BREAD++ is much faster than ARROW while their accuracy is comparable. Hence, we can conclude that BREAD/BREAD++ provides not only fast but also accurate responses and it is of practical value to KR feature extraction in fraud detection system.

Challenging queries simulating fraud feature extraction. Next, we investigate a more challenging task. We simulate the scenario in the real scenario of fraud detection. Let d_{max} be the farthest distance that vertex s can reach in the subgraph G_s, we randomly choose a value d within $[1, d_{max}]$ and set k with a value greater than d for this task. As d_{max} is correlated to the diameter D of the graph and large-scale graphs like Google and WeChat may involve queries with much larger k than previous experiments, this task is more challenging. We generate 1,000 queries for each dataset, consisting of positive and negative queries (ratio is 3:2). The negative queries include (1) target vertices that are outside G_s, and (2) target vertices that are inside G_s but can not be reached under the given hop k. Due to the space limit, we only show the statistics of the hop of generated positive queries on two large graphs Google and WeChat in Fig. 5(a) and Fig. 5(b), respectively. The average running time and the accuracy of each method in such setting is reported in Fig. 3(a) and Fig. 4(a). Observe that ARROW suffers from a significant performance degradation on Flickr, Google and WeChat w.r.t. accuracy. This is consistent with our design of the challenging task: queries on these graphs that have larger diameter come with a larger k than other graphs. Since ARROW chooses k as the walk length of random walks in answering KR queries, it may deviate from its target when performing such longer random walks. On the contrary, the performance of BREAD/BREAD++ remains relatively stable on all datasets.

C. Answering R queries

In our last sets of experiments, we test the effectiveness of BREAD/BREAD++ on the classic reachability problem. We remove the hop constraint of queries used in KR queries and use them as queries for this task. The results are reported in Fig. 3(b) and Fig. 4(b). Similar to the experiments reported in Sec. V-B, BREAD/BREAD++ consistently provides not only fast but also accurate responses than other methods on both the basic task and the challenging task. Moreover, BREAD++ can further improve the efficiency of BREAD while retaining comparable accuracy.

VI. Related Work

A. Reachability on Graphs

Index-based reachability query processing methods can be divided into two categories. The first category applies different data structures (e.g., chains [14], trees [15] and intervals [16]) to compress the complete transitive closure. The second category tries to encode the reachability using a subset of vertices which serve as intermediaries. 2-hop labeling [17] is the most representative technique in this category.

Some works [18], [19], [20], [8] study index-free methods that do not construct indexes for facilitating R queries. Random walk is the prevalent method used for answering index-free R queries. Index-free methods are more flexible to scenarios where maintaining indexes is not feasible or indexes require frequent updates (i.e., dynamic graphs).

There are a few works on answering KR queries [1], [2], [3], [4]. But they are index-based, which is inappropriate for the dynamic graphs such as the social graph of an IMC.

B. Personalized PageRank

Various methods have been proposed in the literature for efficient PPR computation. Fogaras et al. [21] propose the Monte Carlo (MC) approach which conducts a sufficiently large number of random walks to estimate PPR values. Another line of works use the local push algorithm, including Forward Push (for single-source PPR query) [22] and Backward Push (for single-target PPR query) [10], e.g., Zhang et al. [23] extend Forward Push and Backward Push to dynamic graphs, and

Wang et al. [24] design Randomized Backward Search for single-target PPR queries.

There are several works hybridizing MC approach and local push methods to achieve better PPR query efficiency. Lofgren et al. [25], [6] propose FAST-PPR and BiPPR for pairwise PPR computation (i.e., single-source single-target PPR) which adopts Backward Push to reduce the number of random walks that MC requires to estimate $\pi(u, v)$. Wang et al. [26] design HubPPR, an indexing scheme based on BiPPR, for the single-source top-k PPR query. Wang et al. [27], [28] combine Forward Push and MC, and propose FORA, an index-based method, for single-source top-k PPR computation. The Forward Push phase in FORA is further improved by Lin et al. [29]. Wei et al. [30] combine Forward Push, Backward Push and MC to answer single-source top-k PPR queries.

Besides, numerous works parallelize PPR computation in share-memory [12], distributed [31], [32], [33], [34] or GPU-based [35] settings to achieve higher computation efficiency.

VII. Conclusion

In this paper, we study efficiently answering KR queries in large dynamic graphs for fraud feature extraction in WeChat. We propose BREAD/BREAD++ that adopt the idea of estimating PPR value. They have a lower time complexity than existing methods in large-scale dynamic graphs. Experiments on several large graphs including WeChat graph have demonstrated the superiority of our methods. In the future, we plan to exploit more advanced techniques proposed for estimating PPR values in parallel on the CPU or the GPU [12], [11], and further improve our methods.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 62002303), the Natural Science Foundation of Fujian Province of China (No. 2020J05001), the China Fundamental Research Funds for the Central Universities (No. 20720210098), and 2021 Tencent Wechat Rhino-Bird Focused Research Program.

References

[1] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu, "Efficient processing of k-hop reachability queries," *VLDB J.*, vol. 23, no. 2, pp. 227–252, 2014.

[2] X. Xie, X. Yang, X. Wang, H. Jin, D. Wang, and X. Ke, "BFSI-B: an improved k-hop graph reachability queries for cyber-physical systems," *Inf. Fusion*, vol. 38, pp. 35–42, 2017.

[3] M. Du, A. Yang, J. Zhou, X. Tang, Z. Chen, and Y. Zuo, "*HT*: A novel labeling scheme for k-hop reachability queries on dags," *IEEE Access*, vol. 7, pp. 172 110–172 122, 2019.

[4] Y. Cai and W. Zheng, "ESTI: efficient k-hop reachability querying over large general directed graphs," in *DASFAA (Workshops)*, ser. Lecture Notes in Computer Science, vol. 12680, 2021, pp. 71–89.

[5] J. X. Yu and J. Cheng, "Graph reachability queries: A survey," in *Managing and Mining Graph Data*, ser. Advances in Database Systems. Springer, 2010, vol. 40, pp. 181–215.

[6] P. Lofgren, S. Banerjee, and A. Goel, "Personalized pagerank estimation and search: A bidirectional approach," in *WSDM*, 2016, pp. 163–172.

[7] S. Park, W. Lee, B. Choe, and S. Lee, "A survey on personalized pagerank computation algorithms," *IEEE Access*, vol. 7, pp. 163 049–163 062, 2019.

[8] N. Sengupta, A. Bagchi, M. Ramanath, and S. Bedathur, "ARROW: approximating reachability using random walks over web-scale graphs," in *ICDE*, 2019, pp. 470–481.

[9] A. Backurs, L. Roditty, G. Segal, V. V. Williams, and N. Wein, "Towards tight approximation bounds for graph diameter and eccentricities," in *STOC*, 2018, pp. 267–280.

[10] R. Andersen, C. Borgs, J. T. Chayes, J. E. Hopcroft, V. S. Mirrokni, and S. Teng, "Local computation of pagerank contributions," in *WAW*, vol. 4863, 2007, pp. 150–165.

[11] W. Guo, Y. Li, M. Sha, and K. Tan, "Parallel personalized pagerank on dynamic graphs," *Proc. VLDB Endow.*, vol. 11, no. 1, pp. 93–106, 2017.

[12] R. Wang, S. Wang, and X. Zhou, "Parallelizing approximate single-source personalized pagerank queries on shared memory," *VLDB J.*, vol. 28, no. 6, pp. 923–940, 2019.

[13] J. Kunegis, "KONECT: the koblenz network collection," in *WWW*, 2013, pp. 1343–1350.

[14] H. V. Jagadish, "A compression technique to materialize transitive closure," *ACM Trans. Database Syst.*, vol. 15, no. 4, pp. 558–598, 1990.

[15] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD Conference*, 1989, pp. 253–262.

[16] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: scalable reachability index for large graphs," *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 276–284, 2010.

[17] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," in *SODA*, 2002, pp. 937–946.

[18] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovász, and C. Rackoff, "Random walks, universal traversal sequences, and the complexity of maze problems," in *FOCS*, 1979, pp. 218–223.

[19] U. Feige, "A fast randomized LOGSPACE algorithm for graph connectivity," *Theor. Comput. Sci.*, vol. 169, no. 2, pp. 147–160, 1996.

[20] A. Anagnostopoulos, R. Kumar, M. Mahdian, E. Upfal, and F. Vandin, "Algorithms on evolving graphs," in *ITCS*, 2012, pp. 149–160.

[21] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments," *Internet Math.*, vol. 2, no. 3, pp. 333–358, 2005.

[22] R. Andersen, F. R. K. Chung, and K. J. Lang, "Local graph partitioning using pagerank vectors," in *FOCS*, 2006, pp. 475–486.

[23] H. Zhang, P. Lofgren, and A. Goel, "Approximate personalized pagerank on dynamic graphs," in *KDD*, 2016, pp. 1315–1324.

[24] H. Wang, Z. Wei, J. Gan, S. Wang, and Z. Huang, "Personalized pagerank to a target node, revisited," in *KDD*, 2020, pp. 657–667.

[25] P. Lofgren, S. Banerjee, A. Goel, and S. Comandur, "FAST-PPR: scaling personalized pagerank estimation for large graphs," in *KDD*, 2014, pp. 1436–1445.

[26] S. Wang, Y. Tang, X. Xiao, Y. Yang, and Z. Li, "Hubppr: Effective indexing for approximate personalized pagerank," *Proc. VLDB Endow.*, vol. 10, no. 3, pp. 205–216, 2016.

[27] S. Wang, R. Yang, X. Xiao, Z. Wei, and Y. Yang, "FORA: simple and effective approximate single-source personalized pagerank," in *KDD*, 2017, pp. 505–514.

[28] S. Wang, R. Yang, R. Wang, X. Xiao, Z. Wei, W. Lin, Y. Yang, and N. Tang, "Efficient algorithms for approximate single-source personalized pagerank queries," *ACM Trans. Database Syst.*, vol. 44, no. 4, pp. 18:1–18:37, 2019.

[29] D. Lin, R. C. Wong, M. Xie, and V. J. Wei, "Index-free approach with theoretical guarantee for efficient random walk with restart query," in *ICDE*, 2020, pp. 913–924.

[30] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J. Wen, "Topppr: Top-k personalized pagerank queries with precision guarantees on large graphs," in *SIGMOD Conference*, 2018, pp. 441–456.

[31] W. Lin, "Distributed algorithms for fully personalized pagerank on large graphs," in *WWW*, 2019, pp. 1084–1094.

[32] T. Guo, X. Cao, G. Cong, J. Lu, and X. Lin, "Distributed algorithms on exact personalized pagerank," in *SIGMOD Conference*, 2017, pp. 479–494.

[33] S. Luo, "Distributed pagerank computation: An improved theoretical study," in *AAAI*, 2019, pp. 4496–4503.

[34] G. Hou, X. Chen, S. Wang, and Z. Wei, "Massively parallel algorithms for personalized pagerank," *Proc. VLDB Endow.*, vol. 14, no. 9, pp. 1668–1680, 2021.

[35] J. Shi, R. Yang, T. Jin, X. Xiao, and Y. Yang, "Realtime top-k personalized pagerank over large graphs on gpus," *Proc. VLDB Endow.*, vol. 13, no. 1, pp. 15–28, 2019.