Skywalker: Efficient Alias-method-based Graph Sampling and Random Walk on GPUs

Pengyu Wang*, Chao Li*, Jing Wang*, Taolei Wang*, Lu Zhang*, Jingwen Leng*†, Quan Chen*, Minyi Guo*
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
Email: {wpybtw, jing618, sjtuwtl, luzhang}@sjtu.edu.cn, {lichao, leng-jw, chen-quan, guo-my}@cs.sjtu.edu.cn
†Shanghai Qi Zhi Institute, Shanghai, China

***Abstract*—Graph sampling and random walk operations, capturing the structural properties of graphs, are playing an important role today as we cannot directly adopt computing-intensive algorithms on large-scale graphs. Existing system frameworks for these tasks are not only spatially and temporally inefficient, but many also lead to biased results. This paper presents Skywalker, a high-throughput, quality-preserving random walk and sampling framework based on GPUs. Skywalker makes three key contributions: first, it takes the first step to realize efficient biased sampling with the alias method on a GPU. Second, it introduces well-crafted load-balancing techniques to effectively utilize the massive parallelism of GPUs. Third, it accelerates alias table construction and reduce the GPU memory requirement with efficient memory management scheme. We show that Skywalker greatly outperforms the state-of-the-art CPU-based and GPU-based baselines, in a wide spectrum of workload scenarios.**

***Index Terms*—Graph sampling, random walk, graphics processing unit.**

I. Introduction

As a ubiquitous data structure, a graph holds the information of entities and the relationship between them. Classic graph processing algorithms can only capture the low-level features, while traditional machine learning on graphs requires time-consuming feature engineering. In recent years, graph representation learning aims to automatically learn the embedding that encodes the structure information of the graphs for the downstream machine learning tasks. It has shown great promise in recommendation system [1], e-commerce [2], etc.

Graph sampling and random walk algorithms are both important procedures for exploring graphs. The former mainly emphasizes on the local structure while the latter intends to capture more global information. Both types of operations can significantly reduce the ever-growing size of graph data, which allows researchers to adopt deeper and more complicated neural networks on large-scale graphs. Algorithms like `node2vec` [3], `DeepWalk` [4] GraphSAGE [5], ParaGCN [6] and GraphSAINT [7] learn the embedding of nodes or graphs, showing similar or even better results than directly learning from the entire graph. As reported in prior work [8], graph sampling accounts for 31% to 82% of the time for GraphSAGE training. Therefore, the performance of graph sampling is critical for the rapid iteration of GNNs.

Sampling and random walks are considered as embarrassingly parallel computing tasks, which exhibit little communication between walkers or sampling instances. However, effectively leveraging the parallel processors for them is non-trivial. There are two types of sampling and random walks: *unbiased (unweighted)* and *biased (weighted)*. The former one selects neighbors uniformly, and it is generally easy to scale on cores. However, many algorithms use non-uniform sampling probability to select neighbors, hoping to capture more information of the graph structure, termed as *biased* walk/sampling. In contrast, the *biased* version computes the transition probability for neighbors according to edge weights, showing great irregularity due to the nature of graphs.

Since sampling can be seen as a type of graph algorithm, a few researchers have reconfigured the existing general graph processing frameworks to perform sampling and random walk. For example, DrunkardMob [9] provides out-of-memory random walk capability based on GraphChi [10]. Deep Graph Library (DGL) [11] leverages a minimal Gunrock [12] implementation to perform sampling. In general, these systems follow the *vertex-centric* iterative execution model commonly used in graph computing frameworks [12]–[14]. Nonetheless, they lack optimizations specifically for graph sampling and random walks, which exhibit different properties than traditional graph processing workloads.

Specialized system frameworks have been proposed to address the unique characteristics of graph sampling or random walks. For example, there have been studies with CPU-based designs. GraphWalker [15] is a system optimized for random walk on disk-resident graphs, supporting the only unbiased random walk. KnightKing [16] is a distributed system dedicated to random walk algorithms. It adopts the classic *alias method* [17], which is in favor of sampling but requires a preprocessing procedure (building an alias table for all vertices). Another important group of works is to take advantage of the massive parallelism and the high-bandwidth on-board memory of GPUs. Two representative GPU-based sampling frameworks are C-SAW [18] and NextDoor [8]. NextDoor adopts the rejection sampling techniques from KnightKing, but its number of trials is highly dependent on the distribution of biases. C-SAW, on the other hand, uses the *inverse transform sampling* (ITS) [19] method to select vertices, which is costly in terms of time complexity. Besides this, C-SAW neglects high-degree vertices, resulting in biased results.

Based on the above observations, we ask an important question: *can we perform sampling/walking in a spatially and temporally efficient manner while ensuring exact results*?

To answer this question, we thoroughly investigate exiting algorithms, systems, as well as hardware platforms. We found that the alias method has shown great potential. The idea is to jointly exploit the low-time-complexity sampling with the alias method and the massive parallelism with GPU acceleration. Note that the alias method is underestimated and is considered to be not well-suited for running on GPUs in prior works [18], [20]. In this paper, we focused our attention on unleashing the full potential of the alias method on GPUs.

We present Skywalker, an efficient solution for graph sampling and random walks on GPU. We have implemented an easy-to-use programming interface for a variety of algorithms. It includes optimization for unbiased or biased, sampling or random walk algorithms, and real-time or online application scenarios. Skywalker shows significant speedup over the state-of-the-art baseline systems. We are ready to release Skywalker as an open-source project in the near future. To the best of our knowledge, Skywalker is the first system that addresses the challenges of efficiently adopting the alias method for sampling on GPU.

This paper makes the following contributions:

- We introduce a parallel algorithm for alias table construction. To the best of our knowledge, this is the first implementation of the alias method on GPU.
- We introduce *versatile sampler*, a novel execution model for graph sampling and random walk algorithms. It carefully handles the irregularity of graphs and it can reduce the overhead of GPU kernel invoking.
- We introduce efficient buffering techniques using shared memory to accelerate alias table construction. Along with the proposed compressed alias table, the memory requirement of the alias method is greatly reduced.
- We put the above techniques together and present Skywalker, an efficient system for graph sampling and random walk algorithms on GPU. It is heavily optimized for the alias method on GPU in execution and memory efficiency. We show that Skywalker achieves significant speedup over state-of-the-art baselines.

II. Background

A. Graph Sampling and Random Walk

We first introduce the terminology of graph sampling and random walk algorithms. Graph sampling and random walk algorithms are to find a subgraph that can be used to estimate the properties of the original graph.

a) Graph Sampling: Graph sampling algorithms work as follows: For a given graph, one sampler starts from a given root vertex, then repeatedly selects several neighbors of the residing vertex (usually without replacement). Selecting uniformly or according to a given *transition probability* distribution results in unbiased or biased sampling. The probability distribution is often determined by the weight of edges. `Neighbor sampling` samples a constant number of neighbors for each layer. `GraphSage` [5] is an *inductive* algorithm to learn the embedding of a graph using

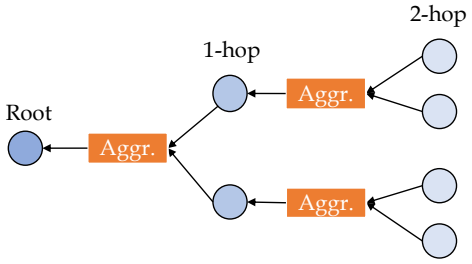


Fig. 1: Graph neural network using `Neighbor Sampling`. Aggregators gather the information of 2-hop neighbors.

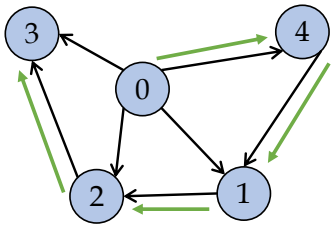


Fig. 2: Random walk on a graph. A walk path is in green.

`Neighbor sampling`. It samples the k-hop neighbors of the root vertices. Figure 1 shows how these GNN algorithms work. `Snow-ball sampling` [21] continually adds newly-discovered neighbors of vertices in the current vertex set until a certain depth. `Forest-fire sampling` [22], [23] is a probabilistic version of `Snow-ball sampling`, following a binomial distribution.

b) Random Walk: The procedure of random walks is similar to sampling. One walker repeatedly selects one neighbor from the residing vertex, and moves itself to the selected vertex until its length reaches a given length. `Personalized PageRank (PPR)` [24], [25] is a sophisticated version of PageRank [26]. It is a biased random walk algorithm with a determination probability for each step. `Deepwalk` [4] is an unbiased walk algorithm for graph embedding. Later work [27] extends `Deepwalk` to a biased version.

The biases in the aforementioned algorithms are static. On the other hand, some algorithms leverage dynamic biases using the runtime information. `node2vec` [3] introduces the 2nd order random walk, defining two hyperparameters p and q to utilize the running states. Specifically, a walker just traveled edge (t, v) and resides at vertex v. Its transition weight to vertex v's neighbor x for the next step is regulated with a parameter α_{pq} as $w'_{vx} = \alpha_{pq}(t, x) \times w_{vx}$, where

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p}, & \text{if } d_{tx} = 0 \\ 1, & \text{if } d_{tx} = 1 \\ \frac{1}{q}, & \text{if } d_{tx} = 2 \end{cases}$$

p is called as the return parameter, deciding the likelihood of immediately revisiting a node.

c) Summary: Random walk algorithms can be seen as special cases of sampling as they both select vertices based on the connectivity of graphs. We use sampling to refer to both graph sampling and random walk algorithms unless otherwise noted. We use *transit vertices* to term the vertices whose

neighbors are to be selected. We use *sample instances* to represent the independent sampling tasks.

B. Key Operations of Sampling

Selecting vertices for unbiased sampling is straightforward as we can generate random numbers from 1 to the degree of the current vertex to select the neighbors. To select vertices for biased sampling, there are several options.

a) Inverse Transform Sampling (ITS): ITS is a common technique for sampling. It computes the cumulative distribution function (CDF) of the bias distribution, and then performs sampling. Suppose a vertex u has neighbors $v_1, v_2, ..., v_n$. The bias of edge E_{uv_i} connecting vertex u and v_i is b_{uv_i}. We can get the CDF as

$$C(j) = \frac{\sum_1^j b_{uv_i}}{\sum_1^n b_{uv_i}}.$$

Given a random number p between 0 and 1, we can find a candidate vertex v_k satisfying that $C(k) \leq p < C(k+1)$. For example, v_1 to v_4, the neighbors of v_0, have bias 1, 1, 1 and 2, respectively. The CDF is computed as Figure 3(a). Suppose we generate the random number as 0.8, then v_4 is the sampled vertex. Computing the CDF requires $O(n)$ time and space while optimized drawing samples once has $O(log\ n)$ complexity using binary search.

b) Rejection Sampling: Rejection sampling does not need to compute the CDF. Instead, it leverages an easy-to-sample distribution to assist the sampling. Specifically, this distribution, called *envelope distribution*, is the upper bound of the bias distribution. For example, the rectangle covering from (0,0) to (4,2), representing a uniform distribution, is an *envelope* of the bias distribution in Figure 3(b). Then we can generate two random numbers to simulate throwing a dart within that envelope. If the height of the sample is lower than the bias of its position, we accept this sample. If the height of the sample is higher than the bias of its position, we reject this sample and repeat this procedure until we find a valid sample. Finally, the accepted samples follow the bias distribution. For more details, please refer to the related textbooks [28].

c) Alias Method: This method constructs two tables, a probability table, and an alias table to draw samples. Specifically, this algorithm assigns the biases of n neighbor vertices into n buckets. The vertices which have biases larger than the average bias distribute their biases to help those vertices with biases lower than the average bias to fill up their buckets. As Figure 3(c) shows, v_1, v_2, v_3 all have a normalized bias 0.8. Thus, v_4 contributes its bias to fill up the bucket of v_1, v_2, v_3. The resulted probability table is $\{0.8, 0.8, 0.8, 1\}$ and the alias table is $\{v_4, v_4, v_4, v_4\}$. To select one sample, we need to generate two random numbers. The first one determines one index while the second one determines to choose the vertex of that index or its alias. Constructing an alias table requires $O(n)$ time and space while the cost of sampling once is $O(1)$.

C. Limitations of the State-of-the-Art

Exiting frameworks all have their limitations. Taking C-SAW (the latest work on GPU-based sampling) for example, it often generates *biased* results. C-SAW leverages prefix-sum to compute the Cumulative Transition Probability Space (CTPS), but it lacks the support for computing the CTPS for extremely high-degree vertices. Its open-sourced implementation [29] just skips the vertices with degrees larger than 8000. In addition, C-SAW uses binary search to draw samples. The time complexity of selecting samples in this way is too high compared with the alias method ($O(log\ n)\ vs.\ O(1)$). It is costly to draw samples for high-degree vertices. Moreover, C-SAW's implementation leverages inflexible data structure for sampled result and intermediate data which is severely limited for high-degree vertices and space inefficient in terms of memory utilization. For example, its in-memory variant can only issue around 4000 random walk instances for Orkut at a time on a single tested GPU with 11 GB memory.

Another representative work on GPU-based sampling is NextDoor [8]. It chooses to leverage rejection sampling similar to KnightKing [16]. However, the average number of trials of rejection sampling is highly dependent on the distribution of biases. For example, Layer Sampling [30] leverages the degree of vertices as bias. For power-law graphs, the biases also have a power-law-like distribution which means that the biases of some vertices are so large that rejection sampling would have a large average number of trials for every trial.

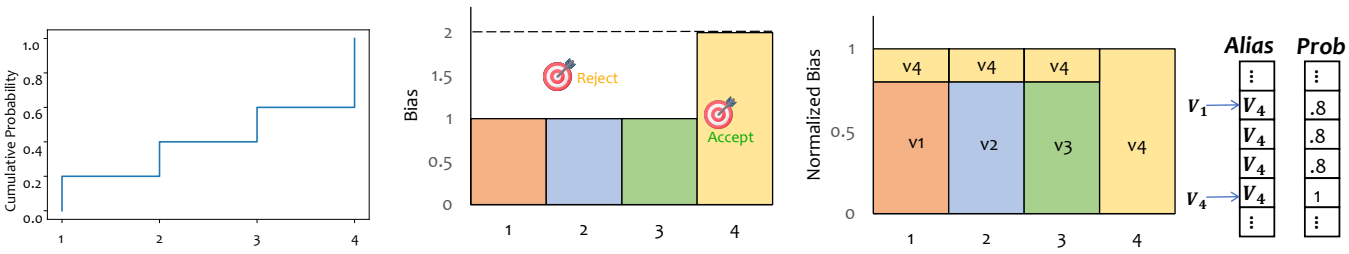
D. Design Opportunities and Challenges

The alias method is preferable in practice as it allows drawing samples in constant time. Once the alias table is constructed, sampling takes constant time as long as the bias does not change. This is preferable as the alias table can be reused across epochs and even different downstream applications. The preprocessing cost can be amortized as the downstream GNN tasks generally need to run tens or hundreds of epochs till convergence. Not to mention that AI researchers often need to run tasks repetitively for network architecture searching [31] and hyperparameter tuning.

Despite the above benefit, the speed for constructing the alias table can be a bottleneck, especially for some algorithms using dynamic biases such as `node2vec`. A natural way to speed up alias table construction is leveraging parallel processing units such as GPGPU. However, constructing an alias table is considered to be non-trivial or problematic on GPUs [18], [20]. Wei et al. [20] found that their GPU implementation for TopPPR [32] is no faster than the multi-thread CPU version.

Specifically, adopting the alias method on GPU faces three challenges:

1) *It is challenging to parallelize the execution of the alias method on GPU.* Constructing the alias table includes a large portion of serial operations. It is non-trivial to map the alias table to GPUs with SIMT execution style. Naively mapping the serial part of the algorithm to each GPU thread would incur *warp divergence* due to unpredictable logical branch, which would severely limit execution efficiency.



(a) Inverse transform sampling (b) Rejection sampling (c) Alias method

Fig. 3: Methods to perform sampling.

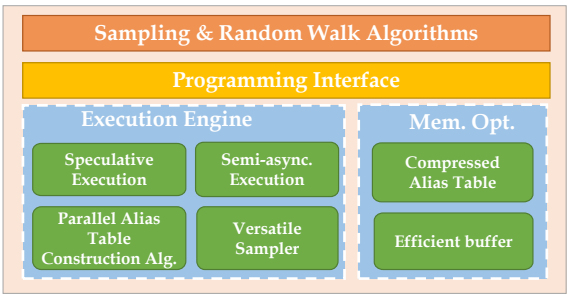


Fig. 4: Skywalker architecture.

2) *It is challenging to balance the parallelism and memory requirement on GPUs.* On one hand, processing many sampling instances concurrently is the key to utilize the massive parallelism of GPUs. On the other hand, constructing an alias table for each vertex requires memory space to storing the necessary intermediate data, which is not insignificant (more details are discussed in §III-C1c). The overall memory requirement of the buffer is proportional to the number of concurrent sampling instances. It is difficult to provide high sampling performance while squeezing the memory budget.
3) *It is challenging to adopt the memory-hungry alias method on resource-limited GPUs.* Besides the memory consumption for the intermediate data, storing the alias method itself requires a larger space than the graph structure data. Thus, the capability of processing large graphs would be greatly limited without sophisticated designs for space efficiency.

III. Skywalker

In this work, we set an ambitious goal of addressing the aforementioned challenges, so that we can enjoy the benefits of the alias method for sampling on GPUs. This section presents Skywalker, our comprehensive solution for graph sampling and random walk on the GPU.

A. System Overview

Skywalker is systemically optimized in three aspects: parallel algorithm, parallel execution engine, and memory optimizations. 1) The parallel algorithm aims to address the first challenge, allowing Skywalker to exploit the parallelism inside each sampling task instance, i.e. *intra-instance parallelism*. 2) The parallel execution engine leverages a novel execution model with multi-level parallel for load-balancing and GPU execution efficiency to fully exploit the parallelism of independent sampling instances, i.e. *inter-instance parallelism*. This execution model allows one to utilize the massive parallelism of GPUs while it does not require much space for intermediate data, addressing the second challenge. 3) Our memory optimizations include the designs to reduce memory requirement so that Skywalker can better handle large graphs, addressing the third challenge.

Skywalker targets both unbiased and biased workloads. Skywalker supports two working modes for biased workloads, *offline* or *realtime* mode. In the offline mode, it constructs the alias table for all vertices in one graph dataset at once as a preprocessing procedure. After the preprocessing, Skywalker performs sampling or random walk directly using the built alias table. In the realtime mode, the alias table of one vertex is constructed for those transit vertices on the fly. Skywalker can cache the constructed alias table for future reuse. For sampling and random walk algorithms using dynamic bias, Skywalker works in the realtime mode. For algorithms using static bias, Skywalker can work in either realtime or offline mode.

B. Exploiting Intra-instance Parallelism

To allow the alias method to effectively run on GPUs, we first parallelize the algorithm to construct the alias table.

a) Parallel Alias Table Construction: We first present a classic serial algorithm, known as Vose's Alias Method [33] in Algorithm 1. Without loss of generality, we first normalize the bias so that the average bias equals to 1. Then, the vertices with biases larger than 1 are inserted into a set $Large$. Correspondingly, the vertices with biases less than 1 are inserted into a set $Small$. Then, v_s and v_l are popped from $Small$ and $Large$. Their biases are used to fill up a bucket of size one. A portion of bias of v_l can help v_s to fill up the bucket of v_s. As the original bias of v_l is larger than 1, its currently remaining bias is larger than 0. If the remaining bias of v_l is larger than 1, v_l is inserted back to $Large$, or $Small$ if the remaining bias of v_l is smaller than 1. This procedure is repeated until $Small$ or $Large$ becomes empty. Note that the resulted alias table is not unique. $Large$ and $Small$ can

be implemented as either stacks or queues as they both lead to valid results.

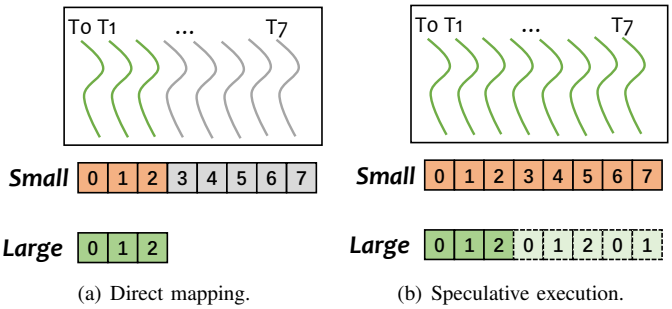
Algorithm 1 Serial algorithm to construct alias table.

```
   Input: B = {b_1, b_2, ...b_n}
   Result: Prob, Alias
1  Large = Small = Alias = ∅;
2  Prob = {p_i | p_i = n*b_i / Σ_1^n b_j, 1 ≤ i ≤ n} ;
3  for i = 1 to n do
4  |  if p_i > 1 then
5  |  |  Large = Large ∪ {v_i};
6  |  else
7  |  |  Small = Small ∪ {v_i};
8  end
9  while Large ≠ ∅ And Small ≠ ∅ do
10 |  v_s = Small.pop();
11 |  v_l = Large.pop();
12 |  Prob[v_l] = Prob[v_l] + Prob[v_s] − 1;
13 |  Alias[v_s] = v_l;
14 |  if Prob[v_l] > 1 then
15 |  |  Large = Large ∪ {v_l};
16 |  else if Prob[v_l] < 1 then
17 |  |  Small = Small ∪ {v_l};
18 end
```

To utilize the parallelism of GPU, Skywalker utilizes multiple threads to compute the alias table of one vertex. Several threads in one thread warp or thread block, termed as a *workgroup*, work cooperatively. Threads in one workgroup can insert vertices into $Large$ and $Small$ simultaneously using atomic operations (line 3 to 8 of Algorithm 1). As for combining the vertices in $Large$ and $Small$ (line 9 to 18 of Algorithm 1), each thread in one workgroup dequeues and processes one pair of large- and small-bias vertices independently as long as there are enough vertices in $Large$ and $Small$.

However, the construction would have limited parallelism if the number of vertices in $Large$ or $Small$ is imbalanced. Consider the example shown in Figure 5(a). This workgroup has eight threads. The size of $Small$ is equal to or larger than 8 while there are only three vertices in $Large$. Thus, five threads will be idle as they can not get one large-bias vertex to process. This situation is common as we cannot assume that $Large$ and $Small$ have similar sizes. Actually, different bias distributions result in $Large$ and $Small$ with different sizes. For example, using vertex degree as the bias on power-law graphs would result in few large-bias vertices and many small-bias vertices. In such cases, many vertices would be idle as they cannot get large-bias to process.

b) Handling Irregular Bias Distribution: To solve the above problem, we propose a technique named *speculative execution*. As the bias of vertices in $Large$ will eventually be distributed into several buckets of vertices in $Small$, we can let several small-bias vertices aggressively consume the bias of large-bias vertices. This could result in negative biases when the biases of those vertices are over-consumed, which is impossible under normal execution. In this situation, those threads that incur over-consumption should roll back their execution. In this way, the workgroup would have full



(a) Direct mapping. (b) Speculative execution.

Fig. 5: Workload mapping strategies have different parallelism when $Large$ and $Small$ are imbalanced.

parallelism even if the size of $Large$ is smaller than the size of $Small$ or the size of the workgroup.

Algorithm 2 Parallel alias table construction.

```
   Input: Large, Small, Prob, Alias
   Result: Prob, Alias
1  while Large ≠ ∅ do
2  |  if ThreadIdx < Large.size then
3  |  |  IsMain = true;
4  |  else
5  |  |  IsMain = false;
6  |  v_s = Small.pop();
7  |  if ThreadIdx == 0 then
8  |  |  Large.size− = MIN(Large.size, GroupSize);
9  |  v_l = Large[ThreadIdx mod Large.size];
10 |  oldP = atomicSub(Prob[v_l], (1 − Prob[v_s]));
11 |  if oldP − (1 − Prob[v_s]) < 0 then    // Roll back
12 |  |  AtomicAdd(Prob[v_l], (1 − Prob[v_s]));
13 |  |  Small = Small ∪ v_s;
14 |  else                                  // Successful update
15 |  |  Alias[v_s] = v_l;
16 |  if IsMain then
17 |  |  if Prob[v_l] > 1 then
18 |  |  |  Large = Large ∪ v_l;
19 |  |  else if Prob[v_l] < 1 then
20 |  |  |  Small = Small ∪ v_l;
21 end
```

As Figure 5(b) shows, eight threads in a workgroup work cooperatively. Each thread has a unique smaller-bias vertex to process. As for large-bias vertices, there are only three of them. Specifically, we term the threads with a local index less than the size of $Large$ as *main threads*. Main threads hold the ownership of the respective vertices, and are responsible for enqueuing that vertex to $Large$ or $Small$ at the end of the steps. Those non-main threads can also consume the bias of one large-bias vertex for its smaller-bias vertices even though they do not hold the ownership. Thus, thread t_0, t_1, t_2 are the main threads, holding the ownership of vertex v_{l_0}, v_{l_1} and v_{l_2}, respectively. All threads try to process their low-bias vertices using atomic operations, speculatively. If the resulted probabilities are valid, the speculative execution is succeeded, and the corresponding threads continue to update the alias table. Otherwise, the corresponding threads withdraw their probability updates. Note that the atomic functions in

Algorithms 2 return the original value before modification, following CUDA's semantic [34]. At the end of this step, main threads enqueue the large-bias vertices to $Large$ or $Small$ based on their current probability. In this way, the parallelism is improved. Algorithms 2 shows the parallel algorithm.

If the workgroup is a block, block synchronization is necessary. For GPUs whose architecture is newer than Volta [35], warp-level synchronization is also needed as the *independent thread scheduling* feature does not guarantee divergent threads in one warp to converge. For most of the vertices, the atomic operations are on shared memory with negligible overhead.

C. Exploiting Inter-instance Parallelism

This subsection presents how Skywalker exploits inter-instance parallelism of graph sampling.

1) Versatile Sampler: Skywalker introduces an execution model named *versatile sampler* for sampling tasks. More than implementing the algorithm introduced in § III-B, versatile sampler allows GPU threads to participate in different levels of collaboration for alias table construction with low overhead.

a) Multi-level Load-balancing: Real-world graphs often have skewed degree distributions, and therefore a onefold parallel execution strategy would cause severely load-imbalance. It can result in stragglers that dominate the execution time. To efficiently handle the skewness of graphs, Skywalker leverages a hierarchical algorithm to assign GPU resources for vertices with varying degrees. Specifically, Skywalker determines the different sizes of workgroups to process one transit vertex using two thresholds: *warp-processing threshold* and *block-processing threshold*. For *low-degree vertices* whose degrees are lower than the warp-processing threshold, 4 or 8 threads in the same warp cooperate to construct the alias table. For *mid-degree vertices* with degrees higher than the warp-processing threshold but lower than the block-processing threshold, Skywalker provides a warp for them. For the remaining *high-degree vertices* with degrees higher than the block-processing threshold, threads in one block work cooperatively.

This strategy is inspired by the *virtual warp-centric* [36] and the *Thread-Warp-CTA* (TWC) [12], [37] strategy used in GPU-based graph processing frameworks. Virtual warp-centric strategy leverages sub-warps to improve the utilization. TWC utilizes thread, warp or CTA (*cooperative thread array*, the same as thread block) to process vertices with varying degrees. But they are different as the operations of traditional graph computing algorithms are generally simpler and independent while constructing the alias table is more complicated and requires cooperation among threads.

b) Role Morphing: Skywalker allows the threads to morph between different roles. Specifically, Skywalker implements subwarp-, warp- and block-collective alias table constructors as GPU device functions. During execution, threads in one block morph between these modes on demand. Similar to the *persistent thread* [38] execution model, the kernel threads which are alive across the execution consume the tasks in a queue until there are no other tasks to be processed. This strategy reduces the kernel invoking overhead. Furthermore, Skywalker leverages shared memory to store the context of samplers. As shared memory has the lifetime of a block [34], the shared memory consumption of the kernel would be the sum of all device functions. To solve this issue, Skywalker statically casts the sampler contexts between subwarp-, warp- and block-collective ones. Note that the threads in one block guarantee the same execution mode by the block barrier.

Unlike the persistent thread model where threads in one block do the same work on different inputs independently, versatile sampler allows threads in one block to morph between three working modes where a different number of threads work collectively to process one task. The load-balancing of the original persistent thread model relies on the assumption that each thread needs to do a similar amount of work for each task. However, the workload of construction the alias table for one vertex, as a basic work unit, is highly varied due to the irregularity of graphs.

c) Memory Requirement Discussion: Alias table construction demands a large size of memory to store the intermediate data. As described above, Skywalker assigns warps and blocks for high- or low-degree vertices. Each SM of an Nvidia GPU can run at most 32 warps concurrently. Thus, it requires thousands of active warps to saturate the hardware resources for a recent GPU with around 100 SMs [39]. Moreover, the sizes of these intermediate data are dependent on the degree of transit vertices, which is highly varied and unpredictable. It is hard to accommodate the memory requirement as real-world graphs may have extremely high degree vertices. Unlike CPU-based frameworks that are free to leverage dynamic arrays, GPU in-kernel memory allocation/deallocation is considered to be slow and unreliable [40], [41]. Even though there are works on dynamic GPU memory allocators [40]–[42], Skywalker chooses to reuse the allocated buffer to fully eliminate the overhead of dynamic memory allocation instead. If we pre-allocate a large buffer capable of storing intermediate data for all processed vertices having the highest degree, the memory requirement would be

$$Mem_1 = K \times \#SM \times WorkgroupPerSm \times MaxDegree$$

where K stands for the memory consumption of constructing an alias table for one element (around 16 bytes). This could consume gigabytes of GPU memory for a graph with 30000 as the highest degree (Orkut, for instance). This would severely limit the capability of processing large graphs.

Skywalker leverages different sizes of workgroups to construct the alias tables. The subwarp- and warp-collective samples are assigned to process the vertices whose degrees have an upper bound. Thus, the memory requirement is reduced to

$$\begin{aligned} Mem_2 = K \times \#SM \times (&SubwarpPerSm \times TH_1 \\ &+ WarpPerSm \times TH_2 \\ &+ BlockPerSm \times MaxDegree) \end{aligned}$$

where TH_1 and TH_2 stands for the warp- and block-processing threshold.

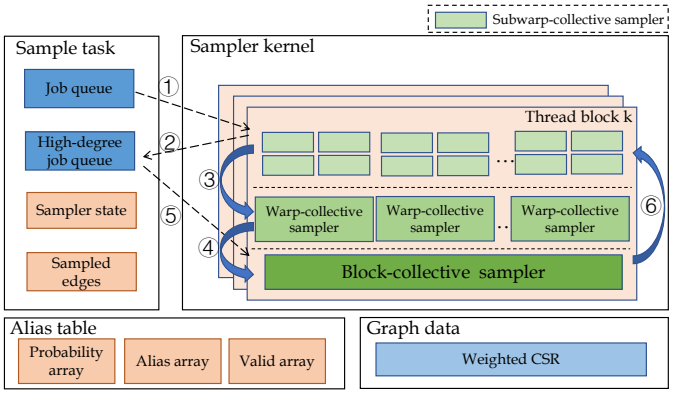


Fig. 6: Skywalker's execution flow.

2) Semi-asynchronous Execution: Sampling systems [9], [18] derived from graph processing systems often follow the synchronous iterative-based execution pattern. However, blocks often have a different amount of workload due to the irregularity of graphs during execution. Those thread blocks processing transit vertices with extremely high degrees could become the stragglers for the current iteration. Moreover, synchronous execution fails to utilize the potential locality existing in sampling algorithms. Sampling one instance only requires the data of k-hop neighbors of the root vertex. Synchronous execution with large batch-size processes transit vertices in the frontier successively, thus would eliminate the locality. Synchronous execution would cause frequent data eviction from the cache to global memory even though such data would be used for the next depth of sampling. For large graphs using streaming processing or Unified Memory (UM), the overhead is even heavier as moving data from the main memory through the PCI-e interface has much higher latency. On the other hand, asynchronous execution is no panacea. Recall that Skywalker leverages warp and block to process collectively. Full-asynchronous execution requires frequent synchronizations within the thread block to convene all threads for collectively processing a high-degree vertex.

To tackle the above problems, Skywalker adopts a *semi-asynchronous* [43] execution strategy for sampling and random walk algorithms. Specifically, the samplers in Skywalker continually request jobs from a per-depth global queue and process its job independently. When one sampler is free and there are no jobs for the current depth in the queue, this sampler advance to process jobs for the next depth without the need of waiting for other samplers.

Figure 6 shows the execution flow of Skywalker: ① Each thread warp runs a subwarp-collective sampler. The samplers acquire jobs from the job queue. If the transit vertex to be sampled has a degree no larger than the warp-processing threshold, the subwarp-collective sampler constructs the alias table, and draws samples for it. ② When a sampler get a high-degree transit vertex, it enqueues that vertex to a queue storing the high-degree jobs. For mid-degree transit vertices versatile sampler temporarily stores them in a per-SM queue. ③ When the global job queue for current iteration is empty, subwarp-collective samplers in one thread warp group together and becomes one warp-collective sampler to process jobs in the per-SM queue. ④ When the per-SM job queue is empty, warp-collective samplers in one thread block group together and becomes one block-collective sampler. ⑤ The block-collective sampler processes the transit vertices in the high-degree queue until empty. ⑥ The block-collective sampler converts back into subwarp-collective samplers for the next iteration.

3) Selecting Vertices: After the construction of the alias table, we want to select vertices based on the alias table. Most of the sampling algorithms adopt *sampling without replacement*. In other words, the selected vertices for one transit vertex should have no redundancy. Skywalker leverages a bitmap for each transit vertex to avoid redundant selection. Specifically, each bit of the bitmap indicates whether one vertex has been selected or not. As GPU cannot directly update the adjacent bits in one byte without using atomic operation, Skywalker uses one byte for each vertex instead. When selecting, threads use the atomic compare-and-swap operation to ensure avoid selecting one vertex repeatedly. As Skywalker adopts the alias method, we can perform resampling with constant overhead. As random walk algorithms select one vertex per depth, there is no need for such a bitmap to avoid redundancy.

For realtime workload, Skywalker selects vertices right after the alias table construction. Several threads in one workgroup select vertices once the workgroup constructs this alias table. For offline workload, Skywalker executes random walk and sampling differently. Specifically, Skywalker follows the semi-asynchronous execution in realtime mode for graph sampling. As one walker selects exactly one vertex per depth, each walker runs on one thread asynchronously till its end.

D. Memory Optimizations

This subsection presents how Skywalker optimizes the memory access and reduces the memory requirement.

1) Accelerating Alias Table Construction with Shared Memory Buffer: To create the probability table and alias table ($Prob$ and $Alias$) for one vertex, we need to load all its neighbors in queues and process them with frequent enqueuing/dequeuing operations. The speed of these operations is crucial for performance. Skywalker leverages shared memory of GPUs to further optimize the buffer. Shared memory, as a region of programmable scratchpad memory inside each SM, provides much lower access latency and higher bandwidth than global memory. Each SM generally has around 48 KB shared memory, which is sufficient to act as a buffer for subwarp- and warp-collective sampler. For example, each SM supports 1024 concurrent threads at most. Thus, each warp can be provisioned with around 1.5 KB shared memory. This means a warp-collective sampler can process nearly 100 elements using buffer shared memory, which is larger than a reasonable warp-processing threshold. For a block-collective sampler, shared memory in each SM alone is not sufficient for processing vertices with extremely high degrees. In this scenario, Skywalker splices shared memory and global memory for the buffer. In

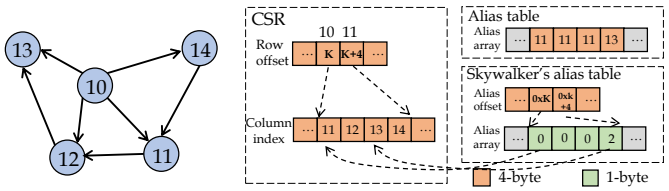


Fig. 7: Skywalker uses the compressed alias table. The probability array is not shown.

other words, the buffer falls back to global memory when the required buffer size is larger than the size of shared memory. Thus, the memory requirement is further reduced to

$$Mem_3 = K \times \#SM \times BlockPerSm \times MaxDegree.$$

This strategy allows to enjoy the low latency of shared memory for the most of time. At this point, Skywalker effectively leverages shared memory on GPU and significantly reduces the memory requirement for alias table construction.

2) Compressed Alias Table: Storing or caching the alias table for future reuse requires large space. The length of the alias array and the probability array of the alias table equals to the out-degree of one vertex. Storing the alias table for the whole graph needs twice space as the graph structure data. Considering the limited memory capacity of GPU, this greatly restricts the capability of sampling large graphs.

To address the above issue, Skywalker efficiently compresses the alias table. Specifically, the original alias method stores the indices of neighbor vertices which range from 1 to $|V|$. For graphs with millions of vertices, a 4-byte format is necessary for identifying the vertex indices. Considering that the degree of vertices in one graph is generally much smaller than $|V|$, Skywalker stores the offset rather than the vertex indices in the alias table. Depending on the maximum degree in on graph, Skywalker can leverage 1-byte, 2-byte, or 4-byte format for the alias array, alternatively.

Recall that the degree of vertices varies and most of the vertices in real-world graphs have low degrees. Skywalker further adopts different formats for different vertices on the alias array. For example, suppose we want to sample a vertex v_{10} with 4 neighbors shown in Figure 7. As the unsigned 1-byte int format can represent numbers up to 255, its alias array can leverage the unsigned 1-byte int format to represent its neighbors with the offset. When drawing samples, the offsets are randomly selected first and are used to find the actual indices of neighbors based on the CSR. To construct such an alias table in a compact format, a preprocessing procedure is necessary. Specifically, Skywalker computes an offset for each vertex to indicate the start position of its alias array. This length of each alias table is computed based on the degree and the format capable of storing the offset of neighbors. The start positions also abide by the alignment requirement. When selecting vertices, Skywalker finds the start position based on the alias offset and then looks up the alias array to find the offset of the selected neighbor in the format determined by its degree. Using the offset, Skywalker gets the neighbor index by looking up the column index array in the CSR.

IV. Evaluation

In this section we evaluate Skywalker. Specifically, we want to answer four questions:

- What kind of speedup can Skywalker bring to unbiased workloads?
- How is the performance of Skywalker on workloads with static or dynamic bias?
- How do the introduced optimizations contribute to the performance?

A. Methodology

a) Platform: We conduct experiments on a Linux server with two 2.40 GHz Intel 20-core, Xeon 6148 CPUs with hyper-threading disabled (40 cores in total). Each CPU has 27.5 MB L3 Cache. The main memory is 256GB. Four NVIDIA RTX 2080Ti GPU with 11GB GDDR6 memory is connected to this system through PCI-e $\times$16 interface. One NVIDIA RTX 2080Ti GPU has 68 multiprocessors (SMs), each with 64 CUDA cores. The operating system is Ubuntu 16.04 with Linux kernel 4.15.0. We use the NVCC compiler version 11.0.167 (g++ version 7.5.0) to compile.

b) Baseline Frameworks: We evaluate the performance of Skywalker comparing with systems:

1) **GraphWalker** [15] is a single-machine graph random walk engine. It preferentially processes the loaded subgraphs to optimize for IO efficiency.
2) **KnightKing** [16] is a distributed graph random walk engine. KnightKing leverages the alias method for algorithms with static bias and rejection sampling for dynamic bias. It proposes two optimizations named outliers handling and pre-acceptance for rejection sampling.
3) **C-SAW** [18] is a GPU-based graph sampling and random walk framework. It leverages the Inverse Transform Sampling method. It computes the cumulative transition probability array using prefix sum. C-SAW also includes optimizations named bipartite region search for resampling, and stridden bitmap for collision detection.
4) **Nextdoor** [8] is a GPU-based framework, utilizing the same rejection sampling technique from KnightKing.

Table I summarizes the baselines and Skywalker. We obtain their source code from Github. Note that GraphWalker only supports unbiased walk and the public APIs of C-SAW do not support `PPR`, `node2vec` and unbiased sampling. Besides these, C-SAW skips all vertices with degrees higher than 8000 as it pre-allocates a fixed-sized buffer for each thread block.

c) Workload: To understand the performance of Skywalker, we carefully select several workloads to cover both sampling and random walk algorithms in different application scenarios. The evaluated algorithms include `Deepwalk`, `PPR`, `node2vec` and `NeighborSampling`. Note that the baseline systems have different processing capabilities and may not support part of the evaluated algorithms. We evaluate both unbiased and biased versions of these algorithms.

TABLE I: Baseline comparison.

Name	Sampling Method	Supported workload
Graphwalker [15]	Unbiased.	Unbiased `PPR`.
KnightKing [16]	Alias method (offline) for static bias, rejection sampling for dynamic bias.	Biased/unbiased random walks.
C-SAW [18]	Inverse transform sampling.	Biased `DeepWalk` and sampling.
NextDoor [8]	Rejection-sampling	Unbiased `node2vec` and `Sampling`. Biased `DeepWalk` and `PPR`.
Skywalker	Alias method (offline and on-the-fly).	Biased/unbiased walks and sampling.

TABLE II: Graph datasets in evaluation from [44], [45]. The sizes are graphs in human-readable weighted edgelist format.

Dataset	Abbr.	$\|V\|$	$\|E\|$	Max Degree	Size(GB)
web-Google	GG	0.9 M	5.1 M	456	0.07
Livejournal	LJ	5 M	69 M	20 K	1.4
Orkut	OK	3 M	117 M	33 K	2.0
Arabic-2005	AB	22 M	640 M	10 K	12
UK-2005	UK	39 M	936 M	5 K	18
Friendster	FS	65 M	1.8 B	3 K	35
SK-2005	SK	50 M	1.9 B	12 K	38

For `NeighborSampling`, we adopt the configuration from GraphSAGE [5] with the sampling depth as 2 and expansion factor (the number of neighbors to be sampled for one vertex) as $S_1 = 25$ and $S_2 = 10$. For `PPR`, we use 15% as determination probability. For `node2vec`, we use hyper-parameter $p = 2.0$ and $q = 0.5$. For all random walk algorithms, we use 100 as the maximum length. For all algorithms, we perform sampling with batch size 40000.

d) Metrics: For most of the experiments, we report the runtime excluding the time of loading data from disk and initialization. For Graphwalker, we exclude the time of repeatedly loading graph chunks during processing. For GPU-based systems, we report the kernel execution time on GPU. We consider the time of alias table construction for the full graph as preprocessing for KnightKing and Skywalker unless specified otherwise. We evaluate the cost of alias table construction for the full graph in § IV-B3. Note that C-SAW skips all vertices with a degree higher than 8000, resulting in much less sampled edges than they should be. We scale the runtime with a factor of the sampled edges.

e) Graph Dataset: We conduct experiments on a variety of widely used datasets listed in Table II. LiveJournal (`LJ`) [44], Orkut (`OK`) [44] and Friendster (`FS`) [44] are social networks. Web-Google (`GG`) [44], Arabic-2005 (`AB`) [45], UK-2005 (`UK`) [45] and SK-2005 (`SK`) [45] are web graph snapshots. We generate edge weight ranging from 1 to 64 uniformly. The size of the datasets varies from 1.4GB to 38GB.

B. Experiment Result

1) Performance of Unbiased Workload.: Table III shows the results of KnightKing, Graphwalker, NextDoor, and Skywalker. Skywalker outperforms the baselines across all test cases. Specifically, Skywalker achieves up to $5894\times$ and $641\times$ average speedup over Graphwalker on `Deepwalk` and `PPR`, respectively. The surprising performance over Graphwalker is because Graphwalker performs random walks on static graph partitions which introduces scheduling overhead.

As for knightking, Skywalker achieves up to $641\times$, $142\times$ and $4157\times$ average speedup over it on `Deepwalk`, `PPR` and `node2vec`, respectively. Note that the speedup of Skywalker on `node2vec` over knightking is larger than other algorithms. This is because `node2vec` algorithm need to check the connectivity of sampled vertices with previously sampled vertex which is time-consuming and the straggler thread would block the execution of all threads. Skywalker does not have this issue as it schedules SMs to work independently.

NextDoor cannot process graphs larger than `UK` due to its high memory requirement. Compared with NextDoor, Skywalker achieves $49.8\times$ and $5.2\times$ average speedup on `node2vec` and `NeighborSampling`.

2) Performance for Biased Workloads.: Figure 8 shows the result of biased workloads. We normalize the result by Skywalker's runtime. Skywalker achieves $3.6\sim21\times$ speedup on `DeepWalk`, $1.7\sim38\times$ speedup on `PPR`, and $2\sim190\times$ speedup on `node2vec` over KnightKing.

As for C-SAW, Skywalker achieves $10\sim93\times$ speedup on `DeepWalk`, $483\sim5878\times$ speedup on `NeighborSampling`. The missing data of C-SAW on graphs larger than AB is because the in-memory version needs to store the full graph in GPU memory. Skywalker does not have this issue as it can utilize the CPU main memory through the Unified Memory mechanism along with space-efficient designs. Besides, the optimized memory management alleviates the overhead caused by oversubscription of Unified Memory. Skywalker outperforms C-SAW so much as Skywalker can utilize the pre-computed alias table while C-SAW always needs to compute the cumulative transition probability. Skywalker shows higher speedup on `NeighborSampling` than `DeepWalk` as Skywalker has lower overhead for sampling multiple items without replacement for two reasons: firstly, *sampling with alias method is more efficient* as it costs constant time while the binary-search approach of C-SAW needs $O(log\ n)$ time. Secondly, *the cost of handling selection collision is lower*. The cost of both collision detection and re-sampling on Skywalker is much lower than on C-SAW.

Comparing with the most recent sampling framework NextDoor, Skywalker achieves $2.6\sim35\times$ speedup on `DeepWalk` and $2.5\sim40\times$ speedup on `PPR`.

3) Construction Alias Table for the Full-graph: Figure 9 compares the runtime of realtime and offline `Deepwalk` on graph `GG` and `OK` with length 100. The runtimes of realtime

TABLE III: Result of unbiased workloads. "_" indicates failures due to internal error. "O.O.M" indicates out-of-memory error.

Workload	Framework	Runtime (ms)							Average Speedup of Skywalker
		GG	LJ	OK	AB	UK	SK	FS	
Deepwalk	Graphwalker	_	172	_	338	721	1719	1739	5894
	Knightking	17	12	13	14	16	3	17	60
	Skywalker	**0.44**	**1.25**	**0.22**	**0.15**	**0.48**	**0.21**	**0.1**	1
PPR	Graphwalker	_	29	_	40	30	73	109	641
	Knightking	3	16	20	16	23	1	16	142
	Skywalker	**0.11**	**0.18**	**0.13**	**0.06**	**0.08**	**0.08**	**0.1**	1
node2vec	Knightking	1189	1382	202	1033	2323	946	122	4157
	Nextdoor	26	38	6.8	17.8	30.4	O.O.M	O.O.M	49.8
	Skywalker	**1.11**	**1.98**	**0.45**	**0.11**	**1.04**	**0.07**	**0.07**	1
Sampling	Nextdoor	1.7	1.97	2	1.7	1.7	O.O.M	O.O.M	5.2
	Skywalker	**0.3**	**0.73**	**1.24**	**0.22**	**0.21**	**0.3**	**0.3**	1

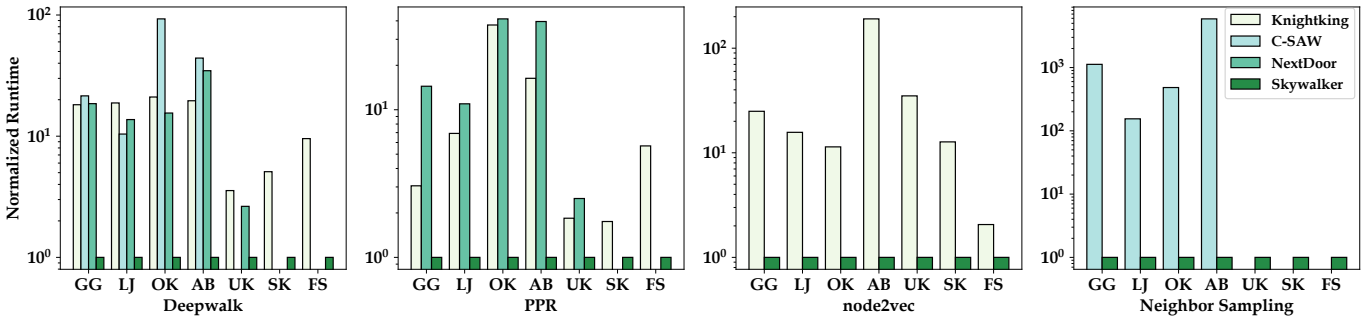
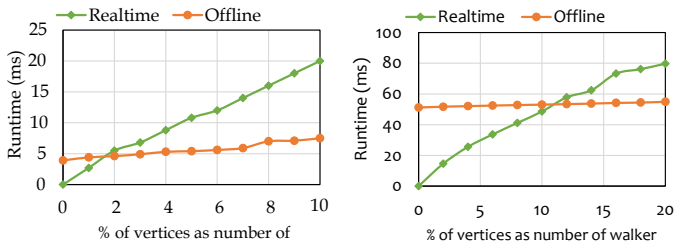


Fig. 8: Results of biased workloads. The runtimes normalized by Skywalker's runtime.



(a) Result on GG. (b) Result on OK.

Fig. 9: The runtime of realtime and online Deepwalk. The x-axis means the number of walkers equal to $|V| \times n\%$.

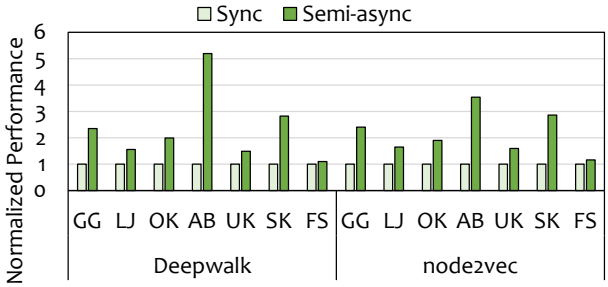


Fig. 10: The normalized performance with synchronous and semi-asynchronous execution strategy.

workload grow proportionally to the number of walkers. For offline workloads, Skywalker constructs the alias table for all the vertices of such graph, and then performs sampling. Invoking walkers with a number of about $2\%|V|$ or $8\%|V|$ have similar runtimes on realtime or offline scenarios for GG and OK, respectively. After such points, the runtimes of offline scenarios grow very gently as Skywalker has significantly higher throughput for offline workloads than realtime workloads. Thus, Skywalker has the flexibility to allow users to choose from realtime mode for the capability of handling dynamic bias or offline mode for higher execution throughput.

4) Speedup Breakdown:

a) Semi-asynchronous Execution: Figure 10 shows the results of Skywalker using its semi-asynchronous or normal synchronous execution strategy. With semi-asynchronous execution, Skywalker achieves 1.1∼5.2× and 1.2∼3.6× speedup on Deepwalk and node2vec, respectively.

b) Speculative Execution: Figure 11 shows the results of SKywalker with or without speculative execution. With speculative execution, Skywalker achieves 1.2∼3.1× and 1.1∼1.6× speedup on Deepwalk and Neighbor sampling, respectively.

c) Compressed Alias Table: Figure 12 shows the space requirement for uncompressed and compressed alias array. The compressed alias array saves over 66% spaces compared with the original version on the evaluated graphs. This allows

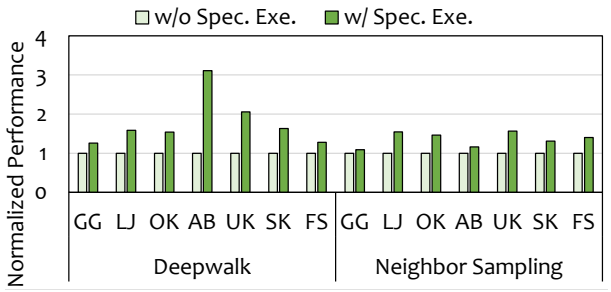


Fig. 11: The normalized performance of Skywalker with and without speculative execution.

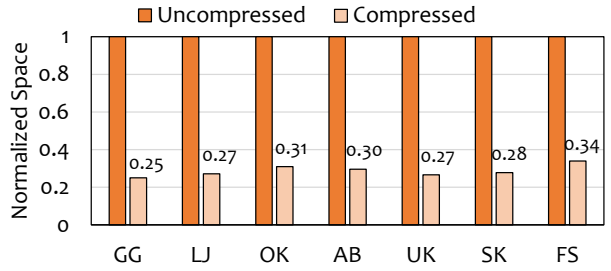


Fig. 12: The normalized space requirement for uncompressed and compressed alias array.

Skywalker to process large graphs, sacrificing no or less overhead of using unified memory or remote GPU memory.

V. Related Works

a) CPU-based Random Walk Systems: DrunkardMob [9] is an out-of-core random walk framework based on GraphChi [10]. It leverages the vertex-centric computational model from graph computing frameworks. KnightKing [16] is a distributed system dedicated to random walk algorithms. It introduces techniques for rejection sampling to assist the alias method. However, these techniques are highly dependent on the distribution of bias. GraphWalker [15] is a recent recently proposed framework while only supports unbiased random walk. Similar to DrunkardMob, it leverages the out-of-core processing capability of GraphChi, and is optimized for IO.

b) GPU-based Sampling Systems: C-SAW [18], a GPU-based framework, supports both graph sampling and random walk algorithms. It leverages a parallel scan algorithm [46] to perform inverse transform sampling. It also optimizes for out-of-memory and multi-GPU sampling, which are unfortunately not shown in its open-source implementation [29]. NextDoor [8] is a GPU-based framework, utilizing the same rejection sampling technique from KnightKing.

c) Algorithm-specific Optimizations: Lin [47] proposed a distributed algorithm for PPR. Bender et al. [48] presented alias algorithms on shared-memory and distributed-memory machines in theory. It includes a straightforward parallel alias algorithm for the shared-memory machines by splitting $Large$ and $Small$ in advance. This cannot be directly adopted on GPU as GPU has different architecture and much higher parallelism that must be explicitly addressed.

d) Handling Large Graphs on GPUs: GTS [49] and Graphie [50] process the graph in the streaming manner. Garaph [51] leverages both the CPU and GPU to collaboratively process the graphs. EtaGraph [52] and Grus [53] adopt Unified Memory to extend GPU memory capacity with the main memory. Subway [54] asynchronously generates the activate subgraphs on the CPU and then accelerates the processing with the GPU.

e) Graph Compression: Besides the general *lossless* and *lossy* data compression techniques [55], [56], graph compression techniques [45], [57], [58] compress the graph structure data. However, these techniques need a time-consuming decompression procedure, which is not suitable for the frequently looked-up alias table. Skywalker does not introduce new general or graph-specific compression or coding techniques. Instead, the alias table can be considered as edge values while values of the alias array are the neighbors. Skywalker exploits the fact that the neighbor indices are also stored in the CSR. By this method, Skywalker can compress the alias array, which allows being looked up with negligible overhead.

f) Summary: This paper attacks the problem of effectively adopting the alias method on GPUs. The proposed parallel algorithm allows to execute the alias method on GPUs effectively. The speculative execution technique explicitly handles the potential irregularity of bias distribution to improve the SIMD efficiency. The execution engine addresses the irregularity of graphs to fully utilize the massive parallelism of GPUs. Besides these, we carefully design the buffer reuse mechanism and compressed alias table to reduce the memory consumption and to improve the large graph processing capability on resource-limited GPUs.

VI. Conclusion

This paper presents Skywalker, a novel graph sampling system that supports a wide variety of unbiased/biased graph sampling and random walk algorithms on GPUs. Specifically, we introduce a parallel algorithm for alias table construction, an efficient parallel execution engine, and a compressed alias table strategy. Skywalker shows significant performance advantage compared to the state-of-the-art baseline systems for a wide spectrum of scenarios including unbiased sampling, unbiased sampling with static and dynamic bias.

Acknowledgment

We thank all the reviewers for their valuable comments and suggestions. This work is supported by the National Natural Science Foundation of China (No.61972247 and No.62072297). Corresponding author is Chao Li from Shanghai Jiao Tong University.

References

[1] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[2] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[3] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[4] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[5] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[6] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Accurate, efficient and scalable graph embedding," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 462–471.

[7] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. K. Prasanna, "Graphsaint: Graph sampling based inductive learning method," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=BJe8pkHFwS

[8] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Nextdoor: Gpu-based graph sampling for graph machine learning," *ArXiv*, vol. abs/2009.06693, 2020.

[9] A. Kyrola, "Drunkardmob: billions of random walks on just a pc," *Proceedings of the 7th ACM conference on Recommender systems*, 2013.

[10] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 31–46. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[11] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[12] Y. Wang, A. A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: a high-performance graph processing library on the GPU," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, 2015, pp. 265–266. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688538

[13] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[14] A. H. N. Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, 2018, pp. 622–636. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173180

[15] R. Wang, Y. Li, H. Xie, Y. Xu, and J. Lui, "Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks," in *USENIX Annual Technical Conference*, 2020.

[16] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "Knightking: a fast distributed graph random walk engine," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, T. Brecht and C. Williamson, Eds. ACM, 2019, pp. 524–537. [Online]. Available: https://doi.org/10.1145/3341301.3359634

[17] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Transactions on Mathematical Software (TOMS)*, vol. 3, no. 3, pp. 253–256, 1977.

[18] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, "C-saw: A framework for graph sampling and random walk on gpus," 2020.

[19] S. Olver and A. Townsend, "Fast inverse transform sampling in one and two dimensions," *arXiv: Numerical Analysis*, 2013.

[20] 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, pp. 15–28, 2019.

[21] A. Stivala, J. Koskinen, D. Rolls, P. Wang, and G. Robins, "Snowball sampling for estimating exponential random graph models for large networks," *Soc. Networks*, vol. 47, pp. 167–188, 2016.

[22] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, ser. KDD '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 177–187. [Online]. Available: https://doi.org/10.1145/1081870.1081893

[23] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 631–636.

[24] D. Fogaras and B. Rácz, "Towards scaling fully personalized pagerank," in *WAW*, 2004.

[25] Q. Liu, Z. Li, J. Lui, and J. Cheng, "Powerwalk: Scalable personalized pagerank via random walks with vertex-centric decomposition," *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking : Bringing order to the web," in *WWW 1999*, 1999.

[27] M. Cochez, P. Ristoski, S. P. Ponzetto, and H. Paulheim, "Biased graph walks for rdf graph embeddings," *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics*, 2017.

[28] D. J. MacKay and D. J. Mac Kay, *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[29] "concept-inversion/c-saw: A framework for graph sampling and random walk on gpus." https://github.com/concept-inversion/C-SAW.

[30] H. Gao, Z. Wang, and S. Ji, "Large-scale learnable graph convolutional networks," *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018.

[31] Y. Gao, H. Yang, P. Zhang, C. Zhou, and Y. Hu, "Graphnas: Graph neural architecture search with reinforcement learning," 2019.

[32] Z. Wei, X. He, X. Xiao, S. Wang, S. Shang, and J.-R. Wen, "Topppr: Top-k personalized pagerank queries with precision guarantees on large graphs," *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[33] M. D. Vose, "A linear algorithm for generating random numbers with a given distribution," *IEEE Trans. Software Eng.*, vol. 17, no. 9, pp. 972–975, 1991. [Online]. Available: https://doi.org/10.1109/32.92917

[34] Nvidia, "Programming Guide :: CUDA Toolkit Documentation," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[35] "Volta Tuning Guide :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/volta-tuning-guide/index.html

[36] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, 2011, pp. 267–276. [Online]. Available: https://doi.org/10.1145/1941553.1941590

[37] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," *SIGPLAN Not.*, vol. 47, no. 8, pp. 117–128, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2370036.2145832

[38] K. Gupta, J. A. Stuart, and J. Owens, "A study of persistent threads style gpu programming for gpgpu workloads," *2012 Innovative Parallel Computing (InPar)*, pp. 1–14, 2012.

[39] "NVIDIA A100 — NVIDIA," https://www.nvidia.com/en-us/data-center/a100/.

[40] X. Huang, C. I. Rodrigues, S. Jones, I. Buck, and W. Hwu, "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines," *2010 10th IEEE International Conference on Computer and Information Technology*, pp. 1134–1139, 2010.

[41] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "Scatteralloc: Massively parallel dynamic memory allocation for the gpu," *2012 Innovative Parallel Computing (InPar)*, pp. 1–10, 2012.

[42] M. Winter, D. Mlakar, M. Parger, and M. Steinberger, "Ouroboros: virtualized queues for dynamic memory management on gpus," *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.

[43] H. T. Kung, "Synchronized and asynchronous parallel algorithms for multiprocessors," *New Directions and Recent Results in. Algorithms and Complexity*, 6 2011.

[44] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, 2012, pp. 745–754. [Online]. Available: https://doi.org/10.1109/ICDM.2012.138

[45] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[46] A. Grimshaw and D. Merrill, "Parallel scan for stream architectures," University of Virginia, Department of Computer Science, Tech. Rep., 2012.

[47] W. Lin, "Distributed algorithms for fully personalized pagerank on large graphs," in *The World Wide Web Conference*, 2019, pp. 1084–1094.

[48] L. Hübschle-Schneider and P. Sanders, "Parallel weighted random sampling," in *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*, ser. LIPIcs, M. A. Bender, O. Svensson, and G. Herman, Eds., vol. 144. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 59:1–59:24. [Online]. Available: https://doi.org/10.4230/LIPIcs.ESA.2019.59

[49] M. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to gpus," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 447–461. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915204

[50] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *26th International Conference on Parallel Architectures and Compilation Techniques, PACT 2017, Portland, OR, USA, September 9-13, 2017*, 2017, pp. 233–245. [Online]. Available: https://doi.org/10.1109/PACT.2017.41

[51] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient gpu-accelerated graph processing on a single machine with balanced replication," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, 2017, pp. 195–207. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ma

[52] P. Wang, L. Zhang, C. Li, and M. Guo, "Excavating the potential of GPU for accelerating graph traversal," in *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, 2019, pp. 221–230. [Online]. Available: https://doi.org/10.1109/IPDPS.2019.00032

[53] P. Wang, J. Wang, C. Li, J. Wang, H. Zhu, and M. Guo, "Grus: Toward unified-memory-efficient high-performance graph processing on gpu," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, Feb. 2021. [Online]. Available: https://doi.org/10.1145/3444844

[54] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: minimizing data transfer during out-of-gpu-memory graph processing," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, 2020, pp. 12:1–12:16. [Online]. Available: https://doi.org/10.1145/3342195.3387537

[55] J. Gilchrist, "Parallel data compression with bzip2," in *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, vol. 16, no. 2004. Citeseer, 2004, pp. 559–564.

[56] S. T. Klein and Y. Wiseman, "Parallel lempel ziv coding," *Discrete Applied Mathematics*, vol. 146, no. 2, pp. 180–191, 2005.

[57] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *Proceedings of the 2008 International Conference on Web Search and Data Mining*, 2008, pp. 95–106.

[58] F. Claude and G. Navarro, "Fast and compact web graph representations," *ACM Transactions on the Web (TWEB)*, vol. 4, no. 4, pp. 1–31, 2010.

APPENDIX

A. Abstract

This artifact contains the source code for Skywalker and shell scripts to set up the environment and perform the evaluation. We describe how to obtain the source code and build the Skywalker project. We illustrate how to download and preprocess the datasets for Skywalker. This artifact also includes the example dataset for baseline systems and Skywalker.

B. Artifact check-list (meta-information)

- **Program:** Biased and unbiased version of `Deepwalk`, `PPR`, `node2vec` and `NeighborSampling`
- **Compilation:** NVCC 11.0 (g++ version 7.5.0), CMake 3.15,
- **Data set:** web-Google , LiveJournal, Orkut, Arabic-2005, UK-2005, Friendster and SK-2005 from SNAP (http://snap.stanford.edu/data/index.html) and Webgraph (http://law.di.unimi.it/datasets.php).
- **Run-time environment:** Ubuntu 16.04 with Linux kernel 4.15.0
- **Hardware:** Turing or newer GPU.
- **Metrics:** Runtime (and sampled edges).
- **Output:** Console and log file.
- **Experiments:** Biased and unbiased version of `Deepwalk`, `PPR`, `node2vec` and `NeighborSampling`. We use batch size as 40000 for most of experiments. The shell scripts to perform evaluation Table 3 and Figure 8 are in `./scripts`.
- **How much disk space required (approximately)?:** 51 GB for Skywalker's dataset. 500 GB for the full evaluation.
- **How much time is needed to prepare workflow (approximately)?:** One hour for compilation. One day for datasets downloading and transformation.
- **How much time is needed to complete experiments (approximately)?:** One day.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache Licenses
- **Data licenses (if publicly available)?:** No
- **Workflow framework used?:** No
- **Archived (provide DOI)?:** Yes. Available at https://doi.org/10.5281/zenodo.5118306

C. Description

1) How to access: A repository that contains the Skywalker code and evaluation scripts can be found in: https://doi.org/10.5281/zenodo.5118306

2) Hardware dependencies: Skywalker requires an NVIDIA GPU with compute capability at least 7.0. An NVIDIA RTX 2080Ti GPU is used to generated sample output.

3) Software dependencies: To compile Skywalker, G++ 7.5, CMake 3.15, and CUDA 10.0 are needed. Later versions might be used not tested by authors.

Skywalker depends on gflags. The baseline systems for comparison include GraphWalker, Knightking, Nextdoor and C-SAW. Please install them following their detailed instructions for a full evaluation. Their repositories are as following:

- https://github.com/rwang067/GraphWalker
- https://github.com/KnightKingWalk/KnightKing
- https://github.com/plasma-umass/NextDoor
- https://github.com/concept-inversion/C-SAW

4) Data sets: Web-Google, LiveJournal, Orkut, Arabic-2005, UK-2005, Friendster and SK-2005 from SNAP, and Webgraph are used in the evaluation. Those datasets need to be transformed into *.gr* format of Galois. Do the following for datasets from SNAP:

```
wget http://snap.stanford.edu/data/wiki-Vote.txt.gz
gzip -d wiki-Vote.txt.gz
$GALOIS_PATH/build/tools/graph-convert/graph-convert
     -edgelist2gr  ~/data/wiki-Vote.txt  ~/data/wiki
    -Vote.gr
```

For datasets from Webgraph, do:

```
wget http://data.law.di.unimi.it/webdata/uk-2005/uk
    -2005.graph
wget http://data.law.di.unimi.it/webdata/uk-2005/uk
    -2005.properties
java -cp "*" it.unimi.dsi.webgraph.ArcListASCIIGraph
        ./uk-2005 ./uk-2005
$GALOIS_PATH/build/tools/graph-convert/graph-convert
     -edgelist2gr  ./uk-2005  ./uk-2005.gr
```

GraphWalker, KnightKing, Nextdoor and C-SAW also require their dedicated data formats. Please follow their instructions to prepare the datasets. Various preprocessed formats of an example dataset (LiveJournal) are included in `dataset`.

D. Installation

- Download artifact from https://doi.org/10.5281/zenodo.5118306 .
- Run `setup.sh`.

E. Experiment workflow

Two shell scripts, `table3_unbiased.sh` and `fig8_biased.sh`, contains the steps to get results for Table 3 and Figure 8 in paper. Edit the install locations of baslines (GraphWalker_DIR, KnightKing_DIR and CSAW_DIR) in `table3_unbiased.sh` and `fig8_biased.sh`. Then, run those scripts to get the results.

F. Evaluation and expected results

Example output files for the experiments are in `./result`.