KnightKing: A Fast Distributed Graph Random Walk Engine

Ke Yang[*] MingXing Zhang[†‡] Kang Chen[*]

Xiaosong Ma[§] Yang Bai[¶] Yong Jiang[†]

Abstract

Random walk on graphs has recently gained immense popularity as a tool for graph data analytics and machine learning. Currently, random walk algorithms are developed as individual implementations and suffer significant performance and scalability problems, especially with the dynamic nature of sophisticated walk strategies.

We present KnightKing, the first general-purpose, distributed graph random walk engine. To address the unique interaction between a static graph and many dynamic walkers, it adopts an intuitive *walker-centric* computation model. The corresponding programming model allows users to easily specify existing or new random walk algorithms, facilitated by a new unified edge transition probability definition that applies across popular known algorithms. With KnightKing, these diverse algorithms benefit from its common distributed random walk execution engine, centered around an innovative rejection-based sampling mechanism that dramatically reduces the cost of higher-order random walk algorithms. Our evaluation confirms that KnightKing brings up to 4 orders of magnitude improvement in executing algorithms that currently can only be afforded with approximation solutions on large graphs.

Keywords graph computing, random walk, rejection sampling

[*]Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China.
[†]Research Institute of Tsinghua University in Shenzhen, China.
[‡]Sangfor Technologies Inc.
[§]Qatar Computing Research Institute, Hamad Bin Khalifa University.
[¶]4Paradigm Co. Ltd.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359634

ACM Reference Format:
Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: A Fast Distributed Graph Random Walk Engine. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3341301.3359634

1 Introduction

Random walk is one fundamental and widely-used graph processing task. As a powerful mathematical tool for extracting information from the ensemble paths between graph entities, it forms a foundation for many important graph measuring, ranking and embedding algorithms, such as personalized PageRank [13, 19, 28], SimRank [21], DeepWalk [34], node2vec [17], among others [10, 12, 20, 27, 28, 33, 35, 36, 42, 46]. These algorithms could work independently, or as a pre-processing step for machine learning tasks [11, 16, 18]. They serve diverse applications, such as node/edge classification, community detection, link prediction, image processing, language modeling, knowledge discovery, similarity measurement, and recommendation.

A random walk based algorithm takes a graph G as input, along with w *walkers*. It starts the walkers, each from a specific vertex, who then wander through the graph independently. At each step, a walker samples an edge from the outgoing edges of its currently residing vertex, following it to the next stop. Each walker quits with a preset termination probability, when reaching a preset path length, or when meeting preset exception criteria. Output can be generated by computation embedded during the random walk process, or by dumping the resulted random walk paths. The process may be repeated for multiple rounds.

Intuitively, the bulk of computation in random walk resides in the edge sampling process, which also embodies the differences between individual random walk algorithms. More specifically, a random walk algorithm defines its own *edge transition probabilities*. As random walk becomes more popular, the sampling logic has also become more complex, with recent algorithms performing *dynamic sampling*, where the edge transition probability depends on the walker's current state, the previous vertex (or vertices) it visited, and the property of edges of its current residing vertex.

As a result, sophisticated random walk algorithms achieve more flexible, application-specific walks, at the cost of sampling complexity. *E.g.*, while the popular network feature

learning technique node2vec [17] includes both dynamic graph random walk and subsequent skip-gram language model construction, it is reported that a Spark node2vec implementation [2] spends 98.8% of its execution time on the former [49]. Our experiment shows, even when implemented above the state-of-the-art graph engine Gemini [50], node2vec is bogged down by edge sampling, producing a vertex navigation rate (number of vertices visited per second) up to 1434 times slower than BFS on the Twitter graph [22].

Such high cost of sampling is mainly due to its dynamic nature: at each step, the out edge selection requires *recalculating all outgoing edges' transition probability*, whose overhead grows with the degree of a walker's current residing vertex. Such overhead is far from balanced across vertices, as real-world graphs tend to have power-law degree distribution [15]. To make things worse, vertices with more incident edges are more likely to be visited, further exacerbating the sampling expense at these hot vertices.

Graphs	Degree mean	Degree variance	Full-scan average overhead	KnightKing's average overhead
Friendster	51.4	1.62E4	361 edges/step	0.77 edges/step
Twitter	70.4	6.42E6	92202 edges/step	0.79 edges/step

Table 1. Node2vec sampling overhead

Table 1 demonstrates this with a side-by-side comparison between two real-world graphs [1] (Twitter [22] and Friendster [47]) with different skewness in edge distribution. They have rather similar average degrees (51 vs. 70), but the Twitter graph is far more skewed, with a variance 395 times higher. Here we report the average sampling overhead of node2vec, in terms of the number of edge transition probabilities computed, per step per walker. Current exact implementations perform full scan of out edges to dynamically calculate their transition probability at each step. This overhead on Twitter is 255 times larger than on Friendster, since the former graph has much more severe imbalance in degrees. It also shows that with such a moderately sized real-world graph (41.7 million vertices and 1.47 billion edges), a dynamic random walk algorithm like node2vec *examines on average nearly 100,000 edges before making each single walker move.*

These challenges apply to many random walk algorithms. However, unlike in the case of graph processing, there lack general-purpose frameworks providing common algorithmic or system support for efficient and scalable graph random walk. As a result, application users develop their own random walk implementations, suffering both redundant labor and poor performance.

Meanwhile, it is counter-intuitive for users to implement these walker-centric algorithms, especially their sampling logic, in popular graph frameworks (such as Pregel [30], PowerGraph [15], Ligra [38], X-stream [37], PowerLyra [9], and Gemini [50]). These graph frameworks, either vertex-centric [9, 15, 30, 38, 50] or edge-centric [37], focus on updating the state of vertices along edges. The "walker" notion central to random walk would likely have to be handled as "messages" with existing systems, losing the capability of tracking/optimizing walker state updates, especially when each walker's next move depends on its recent walk history. Also, many system optimizations adopted by state-of-the-art graph engines, such as 2-D graph partitioning and GAS-like execution, do not fit the random walk computation model and may even backfire to degrade performance, as shown later in the paper.

This paper presents KnightKing, the first general framework for graph random walk. It can be viewed as a "distributed random walk engine", the random walk counterpart of traditional graph engines. KnightKing assumes a *walker-centric* view, with APIs for defining customized edge transition probability, while handling common random walk infrastructure. Similar to graph engines, KnightKing hides system details in graph partitioning, vertex assignment, inter-node communication, and load balancing. Thus it facilitates an intuitive "think-like-a-walker" view, though users have the flexibility to add optional optimizations.

Central to KnightKing's efficiency and scalability is its capability of fast selection of the next edge to follow. We first propose a *unified transition probability definition*, which allows users to intuitively define the *static* (graph-dependent) and *dynamic* (walker-dependent) transition probabilities for custom random walk algorithms. Based on this algorithm definition framework, we build KnightKing as the first random walk system to perform *rejection sampling*. KnightKing eliminates the need of scanning *all* out-edges at a walker's current residing vertex to recalculate their transition probabilities. Intuitively, this step is necessary for dynamic random walk as one cannot proceed with sampling one edge, without evaluating its transition probability relative to its siblings'. With rejection sampling, however, one replaces such complete $O(|E_v|)$ examination at vertex v with only a few trials that are individually and quickly evaluated.

Table 1 demonstrates the difference: for the same node2vec computation, on average KnightKing only needs to compute transition probability for 0.79 edges in sampling one edge. This dramatically speeds up edge sampling, especially with the power-law degree distribution common in real-world graphs: hot vertices with thousands or even millions of edges can be walked in and out at a similar cost as their much less popular peers, saving them from inevitably being frequent, ultra-expensive stops. Also, unlike existing approximate optimizations [2, 49], KnightKing performs *exact sampling*, improving performance without sacrificing correctness.

Finally, though rejection sampling itself is motivated and designed for dynamic sampling, KnightKing is a general-purpose framework and handles static sampling efficiently as well. Besides the above algorithmic innovations, KnightKing encompasses system design choices and optimizations to support its walker-centric programming model.

[1] Both graphs are made undirected for node2vec.

We implemented KnightKing in C++, and evaluated it with four popular random walk algorithms on multiple real-world and synthetic graphs. Results show a speedup up to 4 orders of magnitude for dynamic random walks and 17× for static ones, over fastest alternatives we could find/build. To our knowledge, KnightKing is the only system capable of *exact computation* of dynamic random walk on very large graphs.

2 Unified Walk Algorithm Definition

2.1 Random Walk Taxonomy

We first introduce the common taxonomy of random walk based algorithms. They follow a common framework: given a graph, a certain number of walkers each starts from a given start point, then repeatedly selects a neighbor of the current residing vertex according to given probability distribution, and moves to this neighbor. Intuitively, key variations among random walk algorithms lie in the neighbor selection step.

From the aspect of the edges' relative chances of being selected, random walk algorithms can be categorized into *unbiased*, where the transition probability of out-going edges do not rely on their weight or other properties, and *biased*, where edges are weighted for neighbor selection in walks.

If the edge transition probabilities remain constant throughout the process, we have a *static* random walk. Otherwise, we have a *dynamic* walk, where the determination of transition probabilities involves the walker state, which constantly evolves during a walk. As a result, rather than pre-computing all per-edge transition probabilities, during dynamic walks such probabilities need to be recalculated at each step. We further categorize algorithms in their *order*, by how far back a walker's recent track is considered in updating the transition probability at its current residing vertex. With *first-order* walk algorithms, walkers are oblivious to the vertices visited before the current one, while with *second-order* algorithms, in selecting the next stop, a walker considers the previous vertex visited, from which it transited to the current one. Note that *higher-order* algorithms, including *second-order* ones, are dynamic by definition.

In addition to the transition probability definition, different random walk algorithms might adopt different *termination strategy*. Common strategies include truncating the walk at given number of steps (resulting in walk sequences with uniform length), or having each walker terminate its walk with a given probability at each step.

2.2 Unified Transition Probability

Our literature survey leads us to identify a general framework for defining edge transition probabilities, which applies to known random walk algorithms, across the categories named above. For a walker w currently residing at vertex v, the unnormalized transition probability along its edge e is defined as the product of a *static component* P_s, a *dynamic component* P_d, and an *extension component* P_e. More specifically, the transition probability $P(e)$ is $P_s(e) \cdot P_d(e, v, w) \cdot P_e(v, w)$. Note that the state of w carries necessary history information such as the previous n vertices visited.

Under such a framework, the more naive algorithms are special cases of biased, higher-order algorithms. *E.g.*, an unbiased, static algorithm (which has fixed, uniform transition probability across edges) has both the P_s and P_d components set trivially at 1. Independent of the P_s and P_d definitions, when a walk satisfies the given termination condition (such as reaching a given step count or seeing the "stop" side after flipping a biased coin), P_e becomes 0. Together with cases where no out edges exist or are eligible, a walk terminates as there are no out edges with positive transition probability.

Below we introduce four representative and popular random walk algorithms, and give their edge transition probabilities in the aforementioned format. We focus on the P_s and P_d components. Unless otherwise noted, these algorithms' P_e definition adopts a fixed walk length (80 used in our evaluation, a common setup recommended in prior work [17, 34]).

PPR: A more sophisticated version of the well-known PageRank algorithm is *Personalized PageRank (PPR)* [19, 32]. Unlike the general PageRank problem, which is often computed using power iteration [6], PPR, especially fully PPR (with personalization for all vertices), is known to require prohibitive time or space cost to efficiently compute, especially for large graphs [28]. Random walk based solutions hence become a common approximation, with walk sequences generated and saved for future PPR queries [28]. Our discussion in this paper is based on one random walk implementation using the full personalization model [13, 28], a biased, static algorithm simulating personal preferences in web browsing.

At each step, the probability of an outgoing edge of the current residing vertex being sampled is proportional to its weight. More specifically, $P_s(e) = f(v, x)$, where e connects the current residing vertex v to x, and $P_d(e) = 1, \forall e$. For better performance and parallelism, a long random walk is broken into many short walks, by having walkers adopt a fixed termination probability P_t, so that with P_t, $P_e(v, w) = 0$ ($P_t = 0.0125$ *or* 0.149 in our experiments).

DeepWalk: As another example of biased, static random walk algorithm, consider DeepWalk [34], an important graph embedding technique widely used for machine learning. It leverages language modeling techniques for graph analytics, using truncated random walk to generate many walk sequences. By treating each vertex as a word and each sequence as a sentence, it then applies the SkipGram language model [31] to learn the latent representation of these vertices. This representation is useful in many applications, such as multi-label classification, link prediction, and anomaly detection. While the original DeepWalk was unbiased, later work [10] extends it to biased random walk.

Like with `PPR`, when `DeepWalk` runs on weighted graphs, the transition probability of an edge is also proportional to its

weight. The major difference is then the termination component: unlike PPR, which can have random "early termination", DeepWalk continues till the given path length.

Meta-path: Next we introduce *meta-path based algorithms* [12, 14, 24, 25], proposed to capture the semantics behind the heterogeneity of the vertices and edges. In these algorithms, each walker is associated with a *meta-path scheme* specifying the pattern of edge types in a walk path. *E.g.*, in a graph of publications, to probe the citation relationship, we may set the starting vertex as an author and set the meta-path scheme as "$isAuthor \rightarrow citedBy \rightarrow authoredBy^{-1}$". Longer walks can be created by repeating such templates [12, 40], *e.g.*, in this case generating long citation chains by having alternating edges denoting "cites" (author to paper) and "authored by" (paper to author) relationships.

The actual meta-path to use is application-specific and usually defined by domain experts. Specifically, a Meta-path execution will randomly assign each walker one from N user-supplied meta-path schemes. For a walker assigned a meta-path scheme S, at the k^{th} step:

$$P_d(e) = \begin{cases} 1, & \text{if } type(e) = S_{k \bmod |S|} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

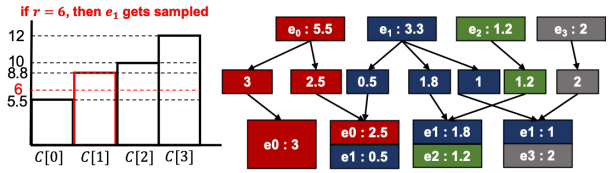
This gives an example of a dynamic, first-order random walk algorithm. At the same vertex, for walkers with different schemes or at different steps, the edge transition probability distribution is different and cannot be pre-computed at the beginning of the execution. Meanwhile, the selection of next edge to follow remains first-order, as it only involves the current position at the assigned scheme, without considering which vertices were visited previously.

node2vec: Finally, we get to introduce a higher-order algorithm. Such algorithms are powerful as the walkers bear their recent walk history in selecting the next stop, which reflects reality in many use scenarios. The vast majority of higher-order algorithms in real applications we have seen so far are 2nd-order [17, 39, 48]. Among them, we introduce the highly popular node2vec [17], which has similar applications as DeepWalk but is more flexible and expressive.

On a given undirected graph, a walker w (that remembers its last stop as $last(w) = t$) has the following dynamic edge transition probability at its current residing vertex v, for edge e, that connects v and vertex x:

$$P_d(e, v, w) = \begin{cases} \frac{1}{p}, & \text{if } d_{tx} = 0 \\ 1, & \text{if } d_{tx} = 1 \\ \frac{1}{q}, & \text{if } d_{tx} = 2 \end{cases} \quad (2)$$

Here p and q are hyperparameters configured by users, and d_{tx} is the distance between t and x. $d_{tx} = 0$ means t and x are the same vertex, which makes $e = (v, x)$ exactly the edge just traveled (*return edge*). Therefore, p is called the *return parameter*, giving the likelihood of immediately revisiting a node in a walk. $d_{tx} = 1$ means x is adjacent with t and $d_{tx} = 2$ otherwise. q thus is called *in-out parameter*,



(a) ITS **(b)** Alias method

Figure 1. Efficient sampling for static walk

where a higher setting generates walks that tend to obtain a "local view" of the underlying graph with respect to the start vertex and approximate BFS behavior. A lower setting, on the other hand, generate walks more inclined to explore nodes "further away", resembling the DFS behavior [17].

node2vec can be either biased or unbiased, as specified by the static component P_s. Commonly, with biased node2vec, P_s and P_e are set by edge weight and fixed walk length, respectively, similar to DeepWalk.

In discussing our proposed solution, we use biased node2vec as a running example, as it illustrates the most complex form of popular random walk algorithms. The less sophisticated algorithms aforementioned can be viewed as special cases of this biased, second-order algorithm, by simplifying part of its edge transition probability definition.

3 Existing Optimizations (Related Work)

Here we summarize existing optimizations for random walk, in part to give background for efficient static sampling, which will be used in KnightKing as well. This section also serves as a brief related work survey.

Static random walk: One common sampling technique for static walks is *Inverse Transform Sampling (ITS)*, illustrated in Figure 1a. Suppose a vertex has n outgoing edges, $\{e_0, e_1, ..., e_{n-1}\}$. An array C can store the *Cumulative Distribution Function (CDF)* of the transition probabilities, by calculating the prefix sum of the unnormalized static component $P_s(e)$, *i.e.*, $C[i] = \sum_{j=0}^{i-1} P_s(e_j)$. The sampling is then performed by generating a random number r in $[0, C[n-1])$ and finding the smallest i where $C[i] > r$ using binary search, producing e_i as the sampled edge. By inversing the "horizontal" sampling to the "vertical" direction, it takes $O(n)$ time and space to construct C, then $O(\log n)$ to sample an edge.

An alternative adopted by existing random walk implementations is the *alias method* [26, 41, 45]. It splits each edge into one or more pieces, with the total number of pieces no larger than $2n$, and put them into n buckets, with the restriction that each bucket holds at most 2 pieces and the sum of weight (P_s) of the pieces in each bucket are exactly the same. These buckets and their content form the alias table. To sample an edge, first we uniformly sample a bucket, then one of its pieces according to their weight, returning the edge the piece belongs to. Here the chance an edge being sampled is proportional to the sum of the weight of its corresponding pieces, which in turn equals its own weight.

Figure 1b illustrates the breaking up of 4 edges, using the same example as above, and their assignment into the 4 buckets. It takes an $O(n)$ pre-processing time and space overhead to build the alias tables, enabling $O(1)$ complexity in edge sampling. KnightKing leverages the alias approach to handle its static transition probability component.

Dynamic random walk: Unfortunately, the aforementioned optimizations do not apply to dynamic random walk, as it brings prohibitive time and space overhead to precompute and store ITS arrays or alias tables enumerating possible walker states. *E.g.*, exact computation of node2vec using CDF or alias requires about 970TB or 1.89PB memory, respectively, on the 11 GB Twitter graph [22]. Consequently, machine learning systems [3–5] implementing such pre-processing for node2vec are known to not scale well [49].

Algorithm-specific optimization, on the other hand, exists for dynamic random walk. *E.g.*, a metapath implementation [1] performs pre-processing to build per-edge-type ITS arrays or alias tables, enabling fast sampling without increasing pre-processing time/space overhead, as edges are partitioned into disjoint sets by type. This, however, cannot be generalized to all dynamic random walks. For node2vec, Fast-Node2Vec [49] uses optimizations such as caching the edge lists of popular vertices to reduce data transmission, and sacrificing the walker processing concurrency to save memory consumption.

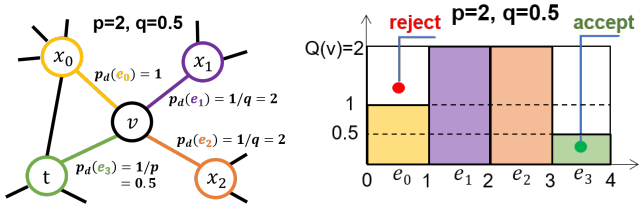
In addition, approximation methods are proposed to make higher-order algorithms more affordable. *E.g.*., node2vec-on-spark [2] trims high degree vertices to enable pre-processing, by selecting only 30 edges for vertices with a higher degree. However, even with this approximation, it still needs to store up to $900|V|$ transition probabilities. Fast-Node2Vec [49] switches to static sampling (by ignoring the dynamic transition probability component) for high-degree vertices.

Unlike these systems, KnightKing enables *exact* edge sampling at $O(1)$ cost, with pre-processing overhead not exceeding the $O(n)$ level, for common random walk algorithms including higher-order ones.

System optimizations: The only system study for random walk we are aware of is DrunkardMob [23], targeting high-speed random walk on multi-core processors. However, it only focuses on static walk and is designed to run out-of-core on a single machine. General-purpose graph computing systems [9, 15, 30, 37, 38, 50] are usually deeply optimized and evaluated for traditional graph algorithms, without addressing random walk workloads.

4 KnightKing Sampling Methodology

This section describes the key innovation within KnightKing: its unified edge sampling mechanism that effortlessly handles expensive dynamic/higher-order walks, while morphing into the alias solution automatically in static walks. Centered around *rejection sampling*, it only needs to compute the unnormalized transition probability of several edges, even at million-edge vertices. Meanwhile, it remains an exact solution, delivering fast sampling without losing correctness.



(a) Sample graph **(b)** Rejection sampling at v

Figure 2. Rejection sampling for unbiased node2vec

4.1 Rejection Sampling for Random Walk

Basic algorithm for unbiased walk: Rejection sampling is originally proposed as a general method for computers to sample from an arbitrary probability distribution [44]. Without loss of generality, we introduce its working through our random walk scenarios.

Consider unbiased node2vec on the sample graph segment shown in Figure 2a. Suppose a walker is currently at v, having previously visited t. As introduced in Section 2.2, the dynamic unnormalized probability component $P_d(e)$ is either $\frac{1}{p}$, 1, or $\frac{1}{q}$, depending on the distance between x and t. With the sample parameter setting of $p = 2$ and $q = 0.5$, the four edges (leading to x_0, x_1, x_2, and t respectively) have P_d valued at 1, 2, 2, and 0.5, respectively. Figure 2b illustrates this with a simple discrete probability distribution plot, with bar heights corresponding to the edges' P_d values.

The basic idea of rejection sampling is to find an *envelope* $Q(v)$ that covers all the bars and convert the 1-D sampling problem among the edges into a 2-D one, within the area covered under the envelope. In KnightKing, we define $Q(v)$ as a per-vertex constant, intuitively drawing it as a horizontal line matching the highest P_d value across all edges. In this case, $Q(v) = max(\frac{1}{p}, 1, \frac{1}{q}) = 2$, as shown in Figure 2b.

To sample one edge, one randomly samples a location (x, y) with uniform distribution within the rectangular area covered by the lines $y = Q(v)$ and $x = |E_v|$, along with the x and y axes. *I.e.*, imagine throwing a dart within that rectangle. The sampled x value gives a *candidate edge* e, while the y value will be compared against its corresponding P_d. If $y \leq P_d(e)$ (the dart hitting a bar), e is *accepted* as a successful sample; otherwise (the dart missing all bars), e is *rejected*, which requires another sampling trial, until success. For sampling correctness, see proof in related textbooks [29].

The beauty of this method lies in that with dynamic random walk, where the transition probability P_d depends on the walker state and cannot be efficiently pre-computed, this rejection-based sampling allows one to sample first, then check whether the sampling can be accepted. This seemingly minor difference eliminates the expensive scan of all edges at the current vertex, needed to update the relative

probability for sampling among the edges. With a reasonable envelope, there is a large chance for the sampling to succeed within a few trials (each with $O(1)$ time to check the actual dynamic probability for *only the sampled edge*). With power-law degree distribution common in real-world graphs, this approach effectively wipes out the difference between vertices, dramatically reducing the sampling cost at vertices with thousands or even millions of edges.

Biased rejection sampling: With our unified definition that decomposes the per-edge unnormalized transition probability into the P_s and P_d components, it is rather straightforward to extend the above basic, unbiased sampling scheme to support biased dynamic random walk.

Again consider node2vec, but with a non-trivial P_s definition (often by the edge weight) to give the static bias to be compounded to the dynamic component P_d. Using existing approaches like ITS and alias (described in Section 3), we can perform optimized 1-D sampling with pre-processing, given the static P_s distribution. Now the candidate edges are sampled according to the P_s values, instead of uniformly. In other words, while in Figure 2b all bars have equal width, here the width of each bar is proportional to the P_s value of the corresponding edge.

Sampling complexity: Another advantage of rejection sampling is that, though conceptually we have the discrete probability distribution plot (Figure 2b), there is no need to physically build such a structure. Given the dynamic component P_d, the per-edge transition probability is computed on-demand using user-defined functions. The static component, on the other hand, benefits from pre-computation given in Section 3, at $O(n)$ time and space for a vertex with n edges, with results reused across all sampling trials.

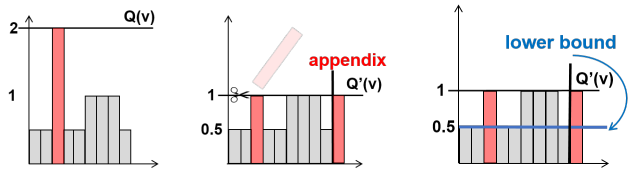
Though individual sampling trials each come with a cost of $O(1)$ (using alias as the static solution) or $O(\log n)$ (using ITS), the overall sampling efficiency apparently depends on the average number of trials needed. Intuitively, the outcome of each trial corresponds to the ratio of "effective area", the combined area of all bars divided by the entire rectangular area. The tighter the envelope bounds the bars, the higher the success rate. Therefore, the average number of trials needed to sample an edge, $\mathbb{E}$ can be calculated as follows:

$$\mathbb{E} = \frac{Q(v) \cdot \sum_{e \in E_v} P_s(e)}{\sum_{e \in E_v} P_s(e) \cdot P_d(e)} \tag{3}$$

Note that this average number $\mathbb{E}$ is not directly related to the degree of v. As a result, with a proper $Q(v)$, $\mathbb{E}$ can be small even with a huge vertex degree, producing dramatic performance gain by reducing the sampling complexity from $O(|E_v|)$ to $\mathbb{E}$ on average.

4.2 Optimization for Dynamic Walk

Handling outliers in P_d: As mentioned, the efficiency of our edge sampling algorithm heavily relies upon how tightly the envelope encloses the per-edge probability bars. The



(a) P_d with outlier **(b)** Outlier optimization **(c)** Global P_d lower bound

Figure 3. Optimizations for dynamic random walk

distribution of P_s is taken care of with the static solution (alias or ITS). However, an undesirable situation remains when P_d has a highly skewed distribution, with a few very tall bars pushing up the entire envelope, as shown in figure 3a. Coming back to the node2vec example, this corresponds to when we have p and q set as 0.5 and 2, respectively. In this case, a single bar (the return edge) has a height of 2, while all the others no taller than 1. This outlier doubles $Q(v)$, creating a rectangular dartboard dominated with invalid white area, which is especially a nightmare when v has many edges.

To mitigate this, one can customize P_d definition to declare such outliers, allowing the system to "fold" them. In the above case, an outlier can be declared for the "return edge" condition, with $P_{outlier}$ value of 2. It is handled by chopping the long bar into 2 parts and appending an "appendix", representing the chopped upper part, to the right of the probability distribution plot. This can be done individually for each outlier case if there are more than one. After generating an x sample, if it falls into the non-outlier area we follow the original rejection sampling scheme. If it falls into an appendix area, we locate the corresponding edge and accept it with a probability of its actual chopped area divided by the estimated appendix area.[2]

Though this step is easy for node2vec, in the worst case one may need to scan all adjacent edges to locate such outlier edges. However, note that we only need to do this when x falls into the appendix area, which is low-probability by definition. Compared with the original "tall" rectangle, the reshaped sampling area makes the average case much faster, at the cost of worst cases when outliers are actually sampled.

Pre-acceptance: A relatively even dynamic probability distribution has high sampling success rate. This actually brings additional opportunities for performance optimization. Note that the envelope $y = Q(v)$ only bounds the y sampling in one direction. With all bars close to the envelope, we reduce another type of waste, by skipping dynamic P_d value check for the sampled candidate edge, which potentially involves expensive process like message passing and remote execution for higher-order algorithms.

Again consider above node2vec example with $1/p = 2$ and $1/q = 0.5$, with all bars above 0.5, as depicted in Figure 3a.

[2]It may be hard to specify the exact width/height of the outliers, without knowing which edges they correspond to for the walker performing this sampling. Users may instead specify upper bounds and subsequently perform corrections in rejection sampling after locating specific outliers.

This way, a dart hitting anywhere below 0.5 is guaranteed to hit a bar, and should be accepted without probability check. Hence we introduce another optimization, by allowing users to provide optional *lower bound* definition. A declaration of lower bound of $L(v)$ notifies the system of another horizontal line $y = L(v)$ (see Figure 3c), allowing it to prune P_d evaluation for samples falling on or below this line.

Our evaluation confirms the effectiveness of these optimizations (Table 5).

5 Workflow and Programming Model

5.1 Random Walk Life Cycle

Similar to traditional graph engines that coordinate updates at many vertices (along many edges) in iterations, KnightKing possess an iterative computation model, coordinating the actions of many walkers simultaneously. Like vertices and edges, walkers are also assigned to each node/thread, based on the assignment of its current residing vertex. The obvious difference from traditional graph engines, of course, is the probabilistic walk along sampled edges vs. deterministic update propagation along all/active edges. A consequent, more subtle difference is on each iteration's execution with a distributed random walk engine.

While graph engines can push/pull updates and perform vertex state updates through one round of vertex-to-vertex messages, when handling higher-order walks KnightKing needs to break such iteration to contain two rounds of message passing. As each walker potentially needs to perform edge selection based on vertices it recently visited, it may issue walker-to-vertex queries (such as in the case of `node2vec`, to check whether the previous stop t is adjacent to a candidate stop x). Under distributed settings, vertices and edges are partitioned across nodes, requiring message passing for sending such queries and collecting their results.

To facilitate efficient batching and coordination of such distributed query operations, KnightKing plays a role similar to a post office, where walkers submit query messages based on their local sampling candidate(s), addressed to vertices involved in dynamic checks. All such queries are delivered according to vertex-to-node assignment, with all nodes work in parallel to process queries received, from all walkers. Then another round of message passing will return results collectively, with querying walkers retrieving results together. Below are the steps within a KnightKing iteration:

1. Walkers generate candidate edges for rejection sampling and perform preliminary screening.
2. Walkers issue walker-to-vertex state queries based on sampling results, when necessary.
3. All nodes process state queries and send back results.
4. Walkers retrieve state query results.
5. Walkers decide sampling outcome, move if successful.

Note that the above describes the most general case of random walk supported by KnightKing. With less complex algorithms, often steps can be skipped. *E.g.*, with static or first-order random walk, steps 2-4 are omitted as there is no need to involve other vertices in local sampling. For such algorithms, all walkers can move lockstep: within each iteration, walkers perform their sampling and (local) checking, until one edge is sampled successfully (or the walk terminates). In these cases, all active walkers are at the same steps along their walk sequences. With higher-order algorithms like `node2vec`, however, iterations across walkers are synchronized by the two rounds of walker-to-vertex query message passing. Lucky walkers with successful sampling will go ahead and walk one step, while less fortunate ones stuck at their current vertex for the next iteration. This way, walkers may proceed at different pace, potentially producing stragglers. We present related optimization in Section 6.2.

```
real_t edgeStaticComp(Edge e) {
  return e.weight;
}
void postStateQuery(Edge e, VertexID src, Walker w) {
  if (w.step != 0) //query w.pre if e.dst is its neighbour
    graph.postNeighbourQuery(w.prev, e.dst, w);
}
real_t edgeDynamicComp(Edge e, Vertex src, Walker w) {
  if (w.step == 0) return max(1.0 / p, 1.0, 1.0 / q);
  else if (e.dst == w.prev) return 1.0 / p;
  else if (graph.getStateQueryResult(w)) return 1.0;
  else return 1.0 / q;
}
real_t dynamicCompUpperBound(EdgeList list, Vertex src) {
  return max(1.0 / p, 1.0, 1.0 / q);
}
real_t dynamicCompLowerBound(EdgeList list, Vertex src) {
  return min(1.0 / p, 1.0, 1.0 / q);
}
```

Figure 4. Node2vec sample code

5.2 KnightKing APIs

Given the general random walk algorithm definition framework proposed in Section 2.1, KnightKing provides intuitive APIs for users to specify existing or create new random walk algorithms. Below we describe its key interfaces.

Transition probability specification: This is the central information users need to supply to KnightKing for implementing a custom random walk algorithm. Recall that the unified transition probability we propose contains the static component $P_s(e)$ and the dynamic one $P_d(e, v, w)$. KnightKing provides two corresponding APIs for their specification, namely, `edgeStaticComp` and `edgeDynamicComp`. Users fill these functions to override the system default of assigning both probability uniformly as 1 across all edges in the graph. Figure 4 gives the sample code of `node2vec` using these APIs.

With `edgeStaticComp`, users provide the static transition probability component, typically as the edge weight (as in the `node2vec` sample code here), or derived from weight or other edge properties. As this definition does not involve the walker state, this component can be pre-computed. Based on this definition, KnightKing performs preparation accordingly during initialization, such as building the alias tables.

`edgeDynamicComp`, on the other hand, provides dynamic computation involving the walker state, which cannot be precomputed. For higher-order algorithms where the dynamic edge transition probability computation involves other vertices, users invoke another interface, `postStateQuery`, to submit walker-to-vertex state queries.

In the `node2vec` example, `edgeDynamicComp` specifies the case-by-case P_d definition, returning 1, $1/p$, or $1/q$ depending on the distance between t (`w.prev`) and x (`e.dst`). To check whether there is an edge between these two vertices, here the `edgeDynamicComp` function further invokes the API `getStateQueryResult`. Note that this is actually executed *after* the two-round message passing for walker-to-vertex state queries, to retrieve query results. The query semantics itself is specified through the `postStateQuery` API. When this function is provided by the user, KnightKing realizes that dynamic walker-to-vertex state checks are necessary and will coordinate the two-round message passing across all walkers. It delivers each such query to the node that hosts the `w.prev` in question, who performs the actual query execution as defined in this function. In this case, user supplied `postStateQuery` calls a standard KnightKing utility routine `postNeighborQuery`, to issue a query checking whether two vertices are neighbors. Beside `postNeighborQuery`, users can also define customized queries.

Finally, the pair of APIs `dynamicCompUpperBound` and `dynamicCompLowerBound` supply the upper and lower bound in P_d for rejection sampling. Like `edgeStaticComp`, these two functions describe static information, thus are invoked by KnightKing during random walk initialization. Note that the former is mandatory for dynamic random walk, to construct the envelope Q. The latter, instead, is an optional API to enable the optimization of reducing dynamic state queries by early acceptance. In this `node2vec` example, the upper bound and lower bound are set in a straightforward way.

Initialization and termination: KnightKing users specify the number of walkers when starting a random walk. They can give specific start locations or their distribution of starting locations (for the system to randomly generate start vertices accordingly) with KnightKing APIs. When not specified either way, start locations are determined using KnightKing's default strategy, which places the i_{th} walker at the $(i \bmod |V|)_{th}$ vertex.

Similarly, KnightKing provides API for users to specify the P_e component by giving termination conditions. For `node2vec`, it is set to continue till reaching 80 steps.

Walker state: KnightKing automatically performs default walker state maintenance, such as updating the current residing vertex, and the number of steps walked. In addition, users performs their own initialization and update of custom walker properties using provided APIs.

Due to space limit, we omit the initialization and termination API usage samples in Figure 4, as well as those for outlier declaration.

6 System Design and Optimization

Finally, we describe system design choices and optimizations within KnightKing, in providing a unified execution engine for diverse random walk algorithms. KnightKing is implemented in around 2500 lines of C++ code, using OpenMPI for inter-node message passing. Its underlying layer has much in common with distributed graph engines, where we adopt infrastructure and techniques in mature systems such as Pregel [30] and Gemini [50]. Our discussion focuses on design aspects and tradeoffs specific to random walk execution.

6.1 Graph Storage and Partitioning

KnightKing store edges using *compressed sparse row (CSR)*, a compact data structure commonly used in graph systems. Further, considering it is important for a walker to directly access any edge of a vertex (*e.g.*, for performing local dynamic transition probability check with rejection sampling), KnightKing adopts a vertex-partitioning scheme. Each vertex is assigned to one node in distributed execution, with all directed edges stored with their source vertices. Undirected edges are stored twice, in both directions.

In general, the frequency of a vertex being visited in a random walk depends on user-defined (and potentially dynamic) transition probabilities and involves intricate interplay between graph topology and walker behavior. In KnightKing, we roughly estimate the amount of processing workload as the sum of a node's local vertex and edge counts, and performs 1-D partition balancing this sum across nodes. While this may not produce evenly distributed random walk processing or communication loads, it achieves even memory consumption, with inadequate memory capacity being the chief cause of distributed processing in the first place.

6.2 Random Walk Execution and Coordination

Computation model: As walkers move independently, random walk algorithms may appear embarrassingly parallel and coordination-free, scaling out easily to more threads/nodes. However, a naive implementation, such as one using a graph database as the storage back-end, with each walker retrieving information via APIs like `getVertex()` and `getEdges()` will incur excessive high network traffic and poor data locality. Instead, KnightKing adopts the *BSP (Bulk Synchronous Parallel)* model [43] common in graph engines, which fits well with its iterative process.

Before a walk starts, KnightKing performs initialization as user specified or by default settings. This includes building per-vertex alias tables if a custom static component P_s is defined, setting up rejection sampling using the upper bound and optional lower bound supplied if a custom dynamic component P_d is defined, and walker instantiation/initialization.

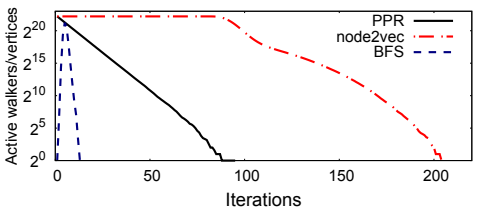


Figure 5. Tail behavior, random walk vs. BFS.

Walker-centric execution of the main walk iterations leverage supports similar to those seen in distributed graph engines: per-walker (rather than per-vertex) message generation, destination node lookup and message batching, and all-to-all message passing. There are also common optimizations such as buffer pool management and pipelining to overlap computation with communication.

Task scheduling: KnightKing performs task scheduling in a similar way as graph engines like Gemini, but works in a walker-centric rather than vertex-centric manner. It sets up parallel processing within each node by having the same number of threads as the number of cores available performing computation, plus two threads dedicated to message passing. The two-round communication within higher-order random walk is implemented within each iteration, with computation (generating outgoing query messages and processing incoming queries) overlapped with message passing. Tasks, defined as chunks of either walkers or messages, are put into shared queues for the threads to grab. The granularity of such dynamic scheduling (chunk size) is set as 128, for both walkers and messages.

Straggler handling: With its rejection-based core strategy reducing the sampling complexity to nearly $O(1)$, KnightKing makes the sampling cost per step dramatically more predictable, not only lower. However, it could still face long-tail executions, with a few stragglers lingering much longer than the bulk of walkers. One scenario leading to this is non-deterministic termination, such as `PPR` (where walkers terminate with a user-set probability). The other is higher-order algorithms like `node2vec`. As mentioned earlier in Section 5.1, with synchronous iterations containing the two-round walker-to-vertex query communication, walkers encountering sample rejection have no choice but stay at where they are to try their luck for the next iteration.

Unlike graph processing algorithms that have to deal with dwindling set of active vertices, here with random walk, we face *longer and thinner* tails, as shown in Figure 5 with the LiveJournal graph. BFS has fast growing and shrinking active vertex set, completing in 12 iterations. In comparison, random walk with stragglers "converges" more slowly, with very few active walkers lagging far behind. As such long tails are algorithm-induced, KnightKing cannot speed up these slow walkers, but we have found that system performance can receive significant improvement by cutting back level of concurrency during this long tail. When there is not much going on, the system overhead in maintaining the original thread pool outweighs the benefit of parallel processing. Therefore a KnightKing node switches to its light mode by retaining only three threads (one for computation and two for message transmission) when its number of active walkers fall below a threshold, set at 4000 in our experiments. Results show that this optimization reduces overall execution time of long-tail walks by up to 57.5% (details in Section 7.5).

7 Evaluation

7.1 Experiment Setup

Testbed: We use an 8-node cluster with 40Gbps IB interconneciton, running Ubuntu 14.04. Each node has 2 8-core Intel Xeon E5-2640 v2, 20MB L3 cache, and 94GB DRAM.

Graph	$\|V\|$	directed $\|E\|$	undirected $\|E\|$	Degree mean	Degree variance
LiveJournal [7]	4.85M	69.0M	86.7M	17.9	2.72E3
Friendster [47]	70.2M	1.81B	3.61B	51.4	1.62E4
Twitter [22]	41.7M	1.47B	2.93B	70.4	6.42E6
UK-Union [8]	134M	5.51B	9.39B	70.3	3.04E6

Table 2. Real-world graph datasets

Input graphs: Table 2 gives specifications of our four real-world graph datasets, widely used in graph processing and random walk evaluations. These graphs cover different graph sizes, with Twitter and UK-Union graphs presenting more pronounced power-law degree distribution than the other two. We use their undirected version, and further create their weighted version (for biased walk) by assigning edge weight as a real number randomly sampled from $[1, 5)$. To be able to manipulate graph scales and characteristics, we also use synthetic graphs, with details to be given in Section 7.3.

Random walk applications: We evaluate four popular random walk algorithms discussed earlier, namely `DeepWalk`, `PPR`, `Meta-path`, and `node2vec`. The first two are static while the last two dynamic. All tests deploy $|V|$ walkers. For `Meta-path`, there are 5 edge types and 10 cyclic path schemes, with length = 5. Each walker is randomly assigned one scheme. For `PPR` we set the termination probability as 1/80, so that its expected walking length is also 80.

Systems for comparison: As there lacks general random walk engines, we compare KnightKing with random-walk-adapted versions of Gemini [50], the state-of-art distributed graph computing system, implemented in C++. It adopts dual-mode (push/pull) update propagation model and performs intensive system optimization targeting graph processing. Like with the GAS model, in Gemini, a vertex cannot directly access all its incident edges, but has to interact with its mirrors distributed on different nodes. So for it we implement a two-phase sampling algorithm: at each step a walker first samples which node to walk to using ITS, then its mirror on that node samples a specific edge. For dynamic walk, the transition probability is computed in an ad-hoc manner, using

ITS also for the second phase (alias inefficient due to its high initialization cost). For static walk, the transition probability and corresponding data structures are pre-computed, with both ITS and alias evaluated for the second phase (results reporting the better between the two). Such vertex replication and edge distribution also prevent Gemini from adopting rejection sampling, as a walker reading any particular edge requires two iterations (sending a request to the mirror and waiting for its response).

Evaluation methodology: Reported execution time (average of 5 runs except several extremely slow cases (*)) includes initializing walkers and sampling related data structures, but excludes graph loading, partitioning, or walking trace collection. With the extremely slow cases, which take the other system evaluated six to hundreds of hours to finish, we extrapolate their performance by running with a small, randomly sampled set of walkers (1% to 6% for Twitter graph and 0.1% to 0.6% for UK-Union). The full execution time is then estimated by using linear regression. As expected, the computation time scales linearly with the number of walkers and the smallest R^2 value in our regression is found to be 0.9998. We further verified our estimation method by completing such a long-running test, where our estimated execution time has an error under 1.5% from the actual one.

7.2 Overall Performance

Time in seconds		Gemini	KnightKing	Speedup $\times times$
DeepWalk	LiveJ	17.64	2.22	7.93
	FriendS	182.09	21.15	8.61
	Twitter	96.90	12.76	7.60
	UK-Union	223.88	38.73	5.78
PPR	LiveJ	110.14	6.50	16.94
	FriendS	297.51	30.82	9.65
	Twitter	201.55	20.27	9.94
	UK-Union	351.99	49.56	7.10
Meta-path	LiveJ	63.59	2.74	23.20
	FriendS	691.07	32.28	21.41
	Twitter	24165*	20.98	1152.03*
	UK-Union	537438*	66.87	8037.50*
node2vec	LiveJ	168.55	14.12	11.93
	FriendS	1467.07	69.80	21.02
	Twitter	97373*	44.14	2206.12*
	UK-Union	1822207*	163.59	11138.85*

Table 3. Overall performance on unweighted graph

Table 3 and Table 4 give overall execution times for the combination of algorithms and input graphs (unweighted and weighted version, respectively). All results are from 8-node executions using 16 threads per node.

With static walk (DeepWalk and PPR), KnightKing executes its unified sampling workflow, but without actually performing rejection sampling. Therefore its performance advantage over Gemini comes from its systems aspects. Overall, KnightKing leads Gemini by up to 16.94× with these two static algorithms 8.22× on average).

Time in seconds		Gemini	KnightKing	Speedup $\times times$
DeepWalk	LiveJ	17.73	3.14	5.65
	FriendS	193.47	30.47	6.35
	Twitter	102.12	17.29	5.91
	UK-Union	233.76	63.24	3.70
PPR	LiveJ	107.52	7.21	14.92
	FriendS	306.10	39.22	7.80
	Twitter	211.99	24.69	8.59
	UK-Union	352.99	70.50	5.01
Meta-path	LiveJ	68.65	3.38	20.32
	FriendS	770.09	47.81	16.25
	Twitter	51783*	30.31	1711.62*
	UK-Union	932536*	97.44	9570.07*
node2vec	LiveJ	170.46	15.34	11.11
	FriendS	1483.33	78.68	18.85
	Twitter	101095*	49.35	2048.53*
	UK-Union	1917205*	189.34	10126.20*

Table 4. Overall performance on weighted graph

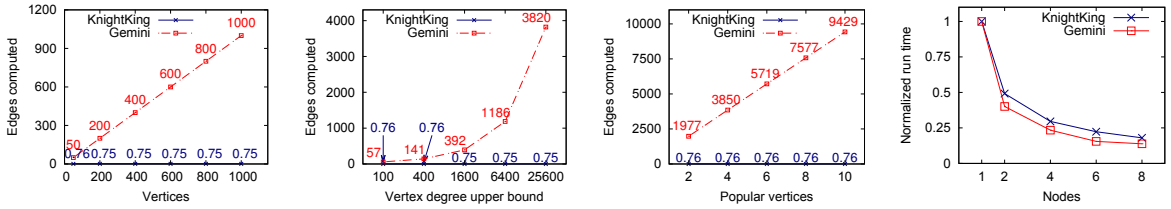
Besides the aforementioned limitation of Gemini's inherent graph partitioning (a vertex accesses its edges through mirrors scattered on different nodes), the performance gap also comes from design mismatch between graph processing systems and graph random walk workloads. *E.g.*, its "dense" mode uses pull-style communication based on in-edges, while walkers naturally work outwards, forcing it to stay with the "sparse" mode. With traditional graph processing, it is common for vertices to update all its neighbors. Therefore Gemini executes such "push" operations via broadcast, for a vertex to notify simultaneously all its mirrors. This introduces huge waste for random walk execution, as a walker only needs to walk along a single edge.

When edges are assigned weights (Table 4), results are similar, with moderate increase of execution time for both systems. That of course comes from the overhead of static sampling with non-uniform transition probability.

Across all input graphs and both systems, PPR is much slower than DeepWalk, due to its non-deterministic behavior. Though its expected walk length is also 80, matching the (fixed) length of DeepWalk, the longest walk length with PPR is over 1000, producing the longer execution time as well as the straggler situation discussed earlier.

When it comes to dynamic algorithms, KnightKing's rejection sampling brings overwhelming advantage. Gemini, using traditional sampling, sees walk execution time grow significantly with Meta-path, and explode with node2vec. For both algorithms, the Twitter graph walk cannot complete within 6 hours, and node2vec on UK-Union is estimated to take >500 hours (with altogether 128 threads on 8 nodes). Also, whether the graph is weighted plays little role for node2vec, due to the dominance of connectivity check cost. These results explain the current adoption of approximation solutions for such higher-order algorithms.

KnightKing, on the other hand, performs exact sampling and reduces such executions with seemingly untameable



(a) Uniform degree **(b)** Truncated power-law degree **(c)** Popular vertices **(a)** Inter-node scalability

Figure 6. Sampling overhead with varying graph topology

Figure 7. node2vec scalability

inherent cost to the time duration close to static algorithms. Across both algorithms and all input graphs (weighted or unweighted), KnightKing finishes within 200 seconds, by reducing the $O(n)$ edge sampling to $O(1)$ level.

7.3 Graph Topology Sensitivity

This section illustrates the impact of moving to rejection sampling, with varied graph topology. Based on the nature of rejection sampling, we are confident about KnightKing's robustness against graph topology changes, including intensifying walk-unfriendly characteristics. The results, therefore, serve as a reminder for what happens to traditional dynamic random walk implementations. For this, we generate synthetic graphs (with 10 million vertices, undirected and unweighted) with different topology features and compare the traditional sampling algorithm (Gemini) with rejection sampling (KnightKing). As we focus on the algorithm aspect here, we report the average number of calculating per-edge transition probabilities needed for walking one step.

First, we use synthetic graphs with uniform degrees, to illustrate the impact of graph density. Figure 6a shows that traditional total sampling overhead growing linearly with vertex degree, while rejection sampling having constant overhead. Note that KnightKing on average computes edge transition probability less than *once* (more accurately, only 0.75 times) per step. This is due to its internal optimizations, such as with the lower-bound-enabled early acceptance.

Next we examine the impact of skewness, by having vertex degrees following a *truncated* power-law distribution found in many real-world graphs. Here "truncated" means there is an upper bound, beyond which the probability density is set to 0. In this setting, the higher the upper bound, the more non-uniform the vertex degree distribution is. As shown in Figure 6b, when the upper bound grows from 100 to 25600, with traditional sampling, the edge probability computation overhead increases by 67 times, while the average degree increases only 3.9 times. KnightKing's rejection sampling unsurprisingly stays constant.

Finally, the existence of hotspots (a few very popular vertices), common in social graphs like Twitter, can drastically slow down random walks. We isolate this behavior by adding a few high-degree vertices (with 1 million edges) to a uniform graph with degree 100. While the behavior of rejection sampling is boring as ever, interestingly the traditional sampling overhead grows linearly with the number of hotspots, as shown in Figure 6c. Adding two such hotspots to the 100-degree uniform graph, one sees the traditional sampling cost skyrocket from 100 to 1977. The more these expensive stops, the higher the average sampling cost per step.

As a by-product of KnightKing's main contribution of lowering the sampling cost to constant level, it makes dynamic random walk resilient to performance hazards brought by treacherous real-world graphs.

7.4 Probability Distribution Sensitivity

		$p=2$ $q=0.5$	$p=0.5$ $q=2$	$p=1$ $q=1$
Exec. time (s)	Naïve	49.22	160.44	43.87
	Lower bound	44.14	145.57	23.53
Edges/step	Naïve	1.05	3.60	1.00
	Lower bound	0.79	2.70	0.00

(a) Impact of lower bound with varied node2vec hyper-parameters

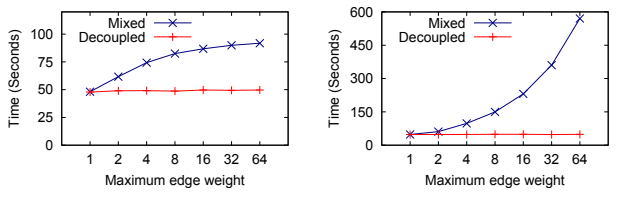
	Naïve	Lower bound (L)	Outlier (O)	L+O
Exec time. (s)	160.44	145.57	84.83	67.21
Edges/step	3.60	2.70	1.81	0.91

(b) Impact of outlier and lower bound optimization with $p=0.5$, $q=2$

Table 5. KnightKing optimizations on node2vec

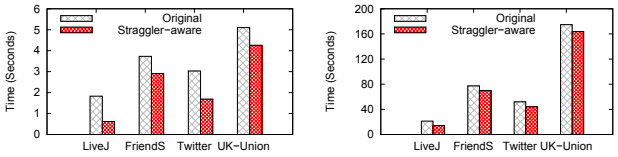
While insensitive to graph topology, KnightKing is sensitive to the P_d distribution (*i.e.*, the "shape" of its dynamic probability distribution plot). Here we evaluate related optimizations to enhance the efficiency of its rejection sampling. Table 5 demonstrates the impact of such optimizations by running unbiased node2vec on the Twitter graph.

In Table 5a, we use 3 sets of node2vec hyper-parameter settings, generating very different run times with naïve rejection sampling. In particular, the 2nd setting $p=0.5, q=2$ has the most skewed P_d, containing a single value of 2 for the return edge, as discussed in Section 4.2. KnightKing is forced to compute the transition probability for 3.6 edges on average in this case. By applying the lower bound optimization to pre-accept edges when the sampled y value is



(a) Uniform **(b)** Power-law

Figure 8. Performance impact of decomposing P_s from P_d



(a) PPR **(b)** node2vec

Figure 9. Impact of scheduling optimization

below the given lower bound, we reduce unnecessary computation and communication, and bring a reduction of up to 10% and 25% for execution time and edges computed per step, respectively, for the first 2 non-trivial cases.

Table 5b fixes the hyper-parameters to the above most challenging setting ($p = 0.5, q = 2$) and shows the effect of outlier mitigation and lower bound optimization, as well as their combination. As expected, the outlier optimization produces significant improvement, almost halving the execution time. Together with lower bound, they bring a 2.4× speed up to the naïve rejection implementation and 75% reduction to the amount of edge probability computation.

The next experiment illustrates that the unified transition probability definition we proposed not only facilitates intuitive and general algorithm specification, but also has performance implications. By decoupling the static (P_s) and dynamic (P_d) components, it isolates the edge weight, often used as the static probability, to one-time static probability pre-computation. Rather than computing the product of P_s and P_d as in traditional dynamic sampling approaches, removing edge weights from P_d reduces its "dynamic range" and subsequently enhances its sampling efficiency.

Figure 8 gives KnightKing performance behavior on `node2vec`, using the Twitter graph, with uniform and power-law weight assignment respectively. We repeat the tests with varied maximum edge weight. With the traditional definition ("mixed"), even with rejection sampling, KnightKing run time grows with the maximum edge weight. Power-law weight assignment, not surprisingly, worsens this growth. By compounding the weight, especially when it has power-law distribution, the overall edge transition probability has higher skewness. This in turn produces more "white area" in the dartboard and slows down rejection sampling. Having separate P_s and P_d ("decoupled"), in contrast, leaves the highly variable weights to static pre-computation and the P_d component more regulated, as reflected by the constant KnightKing run time.

7.5 System Design

We now evaluate the system behavior of KnightKing, starting with scalability. We again show the results with unbiased `node2vec`, which as a 2nd-order algorithm, involves far more inter-node communication than other algorithms. Figure 7a shows the comparison with Gemini on the Friendster graph, using growing number of nodes. Both systems scale quite similarly, though not linearly (to be expected with such irregular computation). Note that results are normalized to each system's single-node run time, and the KnightKing baseline has a 20.9× advantage over Gemini's.

Finally, we assess KnightKing's straggler-aware scheduling (Section 6.2), on the two straggler-prone algorithms `PPR` (P_t set as 0.149 [28]) and `node2vec`. Figure 9 gives the KnightKing run times on the three input graphs. Here the baseline performance, using the original scheduler, includes the outlier and lower bound optimizations, which also alleviate the long-tail situations. Nevertheless, simply by reducing the number of threads to 3 on a node when there are under 4000 active walkers there, this straggler-aware optimization helps the two algorithms reduce execution time by up to 66.1% (on average 37.2% for `PPR` and 16.3% for `node2vec`). Its brings more improvement to small graphs (LiveJournal in this case), as with overall less work to do, the long-tail part of the execution has higher effect on performance.

8 Conclusion

This work presents KnightKing, to our knowledge the first general-purpose, distributed graph random walk engine. It provides an intuitive walker-centric computation model to support easy specification of random walk algorithms. We propose a unified edge transition probability definition that applies across popular known algorithms, and novel rejection-based sampling schemes that dramatically reduce the cost of expensive higher-order random walk algorithms. Our design and evaluation of KnightKing demonstrate that it is possible to achieve near $O(1)$ complexity in *exact* edge sampling, regardless of the number of outgoing edges at the current vertex, without losing accuracy.

ACKNOWLEDGMENT

We thank our shepherd and the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by National Key Research & Development Program of China (2018YFB1003505), National Natural Science Foundation of China (61433008, 61373145, 61572280), Young Scientists Fund of the National Natural Science Foundation of China (61802219), China Postdoctoral Science Foundation (2018M630162). Corresponding Author: Kang Chen (chenkang@tsinghua.edu.cn).

References

[1] [n.d.]. Euler: a distributed graph deep learning framework. https://github.com/alibaba/euler.

[2] [n.d.]. Node2vec on Spark. https://github.com/aditya-grover/node2vec.

[3] [n.d.]. Node2vec with tensorflow. https://github.com/apple2373/node2vec.

[4] [n.d.]. OpenNE: An open source toolkit for Network Embedding. https://github.com/thunlp/OpenNE.

[5] [n.d.]. Stanford Network Analysis Platform (SNAP). https://github.com/snap-stanford/snap.

[6] Arvind Arasu, Jasmine Novak, Andrew Tomkins, and John Tomlin. 2002. PageRank computation and the structure of the web: Experiments and algorithms. In *Proceedings of the Eleventh International World Wide Web Conference, Poster Track*. 107–117.

[7] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 44–54.

[8] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems*. ACM, 1.

[10] Michael Cochez, Petar Ristoski, Simone Paolo Ponzetto, and Heiko Paulheim. 2017. Biased graph walks for RDF graph embeddings. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics*. 21.

[11] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. 2018. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering* (2018).

[12] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 135–144.

[13] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.

[14] Tao-yang Fu, Wang-Chien Lee, and Zhen Lei. 2017. Hin2vec: Explore meta-paths in heterogeneous information networks for representation learning. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. ACM, 1797–1806.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *the Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA, 17–30.

[16] Palash Goyal and Emilio Ferrara. 2018. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems* 151 (2018), 78–94.

[17] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 855–864.

[18] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017).

[19] Taher H Haveliwala. 2002. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*. 517–526.

[20] Mohsen Jamali and Martin Ester. 2009. Trustwalker: a random walk model for combining trust-based and item-based recommendation. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 397–406.

[21] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 538–543.

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World Wide Web*. ACM, 591–600.

[23] Aapo Kyrola. 2013. Drunkardmob: billions of random walks on just a pc. In *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 257–264.

[24] Ni Lao and William W Cohen. 2010. Fast query execution for retrieval models based on path-constrained random walks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 881–888.

[25] Sangkeun Lee, Sungchan Park, Minsuk Kahng, and Sang-goo Lee. 2012. Pathrank: a novel node ranking measure on a heterogeneous graph for recommender systems. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 1637–1641.

[26] Aaron Q Li, Amr Ahmed, Sujith Ravi, and Alexander J Smola. 2014. Reducing the sampling complexity of topic models. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 891–900.

[27] Juzheng Li, Jun Zhu, and Bo Zhang. 2016. Discriminative deep random walk for network classification. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 1004–1013.

[28] Qin Liu, Zhenguo Li, John Lui, and Jiefeng Cheng. 2016. Powerwalk: Scalable personalized pagerank via random walks with vertex-centric decomposition. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 195–204.

[29] David JC MacKay and David JC Mac Kay. 2003. *Information theory, inference and learning algorithms*. Cambridge university press.

[30] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[31] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *International Conference on Learning Representations (2013)*. 1–12.

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[33] Shirui Pan, Jia Wu, Xingquan Zhu, Chengqi Zhang, and Yang Wang. 2016. Tri-party deep network representation. *Network* 11, 9 (2016), 12.

[34] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.

[35] Bryan Perozzi, Vivek Kulkarni, Haochen Chen, and Steven Skiena. 2017. Don't Walk, Skip!: Online Learning of Multi-scale Network Embeddings. In *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*. 258–265.

[36] Leonardo FR Ribeiro, Pedro HP Saverese, and Daniel R Figueiredo. 2017. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 385–394.

[37] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles(SOSP2013)*. ACM, 472–488.

[38] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*,

Vol. 48. ACM, 135–146.

[39] Guolei Sun and Xiangliang Zhang. 2017. Graph Embedding with Rich Information through Heterogeneous Network. *arXiv preprint arXiv:1710.06879* (2017).

[40] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment* 4, 11 (2011), 992–1003.

[41] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*. 1067–1077.

[42] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast random walk with restart and its applications. In *Sixth International Conference on Data Mining (ICDM'06)*. 613–622.

[43] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[44] John Von Neumann. 1951. Various techniques used in connection with random digits,. *Applied Math Series* 12, 36-38 (1951), 5.

[45] Alastair J Walker. 1977. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)* 3, 3 (1977), 253–256.

[46] Yubao Wu, Yuchen Bian, and Xiang Zhang. 2016. Remember where you came from: on the second-order random walk based proximity measures. *Proceedings of the VLDB Endowment* 10, 1 (2016), 13–24.

[47] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[48] Ziqian Zeng, Xin Liu, and Yangqiu Song. 2018. Biased Random Walk based Social Regularization for Word Embeddings. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. 4560–4566.

[49] Dongyan Zhou, Songjie Niu, and Shimin Chen. 2018. Efficient Graph Computation for Node2Vec. *arXiv preprint arXiv:1805.00280* (2018).

[50] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *the Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 301–316.