TEA: A General-Purpose Temporal Graph Random Walk Engine

Chengying Huan[1,2], Shuaiwen Leon Song[3], Santosh Pandey[4], Hang Liu[4], Yongchao Liu[5], Baptiste Lepers[6], Changhua He[5], Kang Chen[1], Jinlei Jiang[1], Yongwei Wu[1]
[1]Tsinghua University; [2]Baihai Technology Inc.; [3]University of Sydney; [4]Stevens Institute of Technology; [5]Ant Group; [6]Université de Neuchâtel

Abstract

Many real-world graphs are *temporal* in nature, where the temporal information indicates when a particular edge is changed (e.g., edge insertion and deletion). Performing random walks on such temporal graphs is of paramount value. The state-of-the-art sampling strategies are tailored for conventional static graphs and thus cannot effectively tackle the dynamic nature of temporal graphs due to several significant efficiency challenges, i.e., high sampling complexity, gigantic index space, and poor programmability.

In this paper, we present *TEA*, the first highly-efficient general-purpose TEmporal grAph random walk engine. At its core, TEA introduces a new hybrid sampling approach that combines two Monte Carlo sampling methods together to drastically reduce space complexity and achieve high sampling speed. TEA further employs a series of algorithmic and system-level optimizations to remarkably improve the sampling efficiency, as well as provide streaming graph support. Finally, we introduce a temporal-centric programming model to ease the implementation of various random walk algorithms on temporal graphs. Experimental results demonstrate that TEA can achieve up to *3 orders of magnitude* speedups over the state-of-the-art random walk engines on large temporal graphs.

CCS Concepts: **• Computing methodologies → Parallel algorithms**; **• Theory of computation → Graph algorithms analysis**.

Keywords: Random walk; Graph algorithm; Temporal graph

ACM Reference Format:
Chengying Huan[1,2], Shuaiwen Leon Song[3], Santosh Pandey[4], Hang Liu[4], Yongchao Liu[5],, Baptiste Lepers[6], Changhua He[5], Kang Chen[1], Jinlei Jiang[1], Yongwei Wu[1]. 2023. *TEA*: A General-Purpose Temporal Graph Random Walk Engine. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3567491

1 Introduction

Many real-world graphs are *temporal* in nature, where the temporal information indicates when a particular edge is changed (e.g., edge insertion and deletion). And such temporal information is often crucial to correctly interpreting temporal graphs. Figure 1 uses the commuting network to demonstrate the importance of temporal information. In the commuting graph, if a path is to be formed, the path must obey the temporal connectivity rule. That is, passing through each vertex, the time of the out edge from this vertex is larger than that of the in edges. Using the paths arriving at vertex 7 from vertex 9 as an example, only three paths 9⟶7⟶4, 9⟶7⟶5, and 9⟶7⟶6 are valid. Apparently, this is different from the case when we disregard the temporal information.

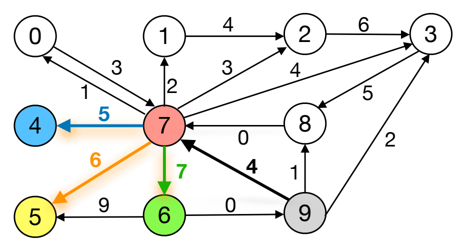


Figure 1. Commuting network represented as a temporal graph where the numerical value on each edge represents the departing time from the source to the destination vertex. This running example is used across this manuscript.

For many real-world applications, temporal information is a key metric for extracting valuable insights and making informed decisions. Below, we enumerate a few more examples, in addition to the aforementioned commute network. In an e-commerce network [22, 51], users' preferences could evolve from time to time. Static graph analysis would overlook such information and result in inaccurate or misleading market decisions, leading to severe revenue losses. Another example is an education network [22]. A student who has not attended class in the last few days will have a higher

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05...$15.00
https://doi.org/10.1145/3552326.3567491

probability of dropping out. Educators could intervene proactively to avoid such abrupt career changes. The temporal information in a temporal graph is often indispensable.

Random walk is a popular and fundamental tool for many graph applications like graph processing, link prediction, graph mining, graph embedding, and node classification [3–5, 7, 8, 10, 33, 37, 45, 47–53]. In general, a random walk usually starts from a specific vertex. At each step, this walker samples an edge from the outgoing neighbors of its currently residing vertex according to the *transition probability* defined by each random walk application [8, 33, 45]. This process will continue until it meets certain termination criteria, such as the desired random walk length. Several recent efforts [32, 40, 45] have developed systems to support random walks and their applications on *static graphs* which disregard the temporal information. However, various graph learning projects [17, 25, 31, 41, 55] identify that integrating temporal information into random walks can dramatically improve graph learning accuracy, demonstrating the importance of temporal random walks.

Unlike a static graph, a temporal graph walker must guarantee that *the time order of a path increases*. Specifically, a temporal graph walker starts from a specific edge. At each step, this walker samples an edge, which has a larger time instance than the current edge, from the out-edges of its currently residing vertex according to the *transition probability*. In static graphs, the edge sampling step is deemed as the most challenging step for a random walk algorithm [32, 35, 40, 45]. When it comes to temporal graphs, the additional temporal information will, unfortunately, further complicate that sampling process.

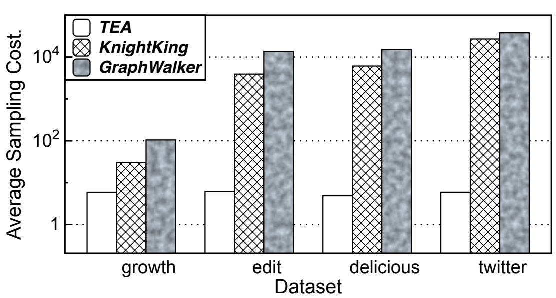


Figure 2. Average sampling cost of *TEA* (hybrid sampling), KnightKing (rejection sampling), and GraphWalker (full-scan sampling), which is defined as #edges/step.

First, rejection sampling, regarded as a desirable sampling algorithm for dynamic random walks according to [45], suffers from a high rejection rate in temporal random walks. Particularly, in temporal random walk algorithms, the edge weight is often associated with temporal information. For instance, an exponential temporal random walk uses the exponential value of the temporal information to represent the sampling bias (Section 2.3). This will result in a highly skewed probability distribution function (Section 2.2), which leads to a drastically squeezed "accept" area. Therefore, *rejection sampling* [39] method will suffer from a large number of average trials due to high rejection rate. Consequently, one has to resort to either *inverse transform sampling (ITS)* method [30] or the alias method for sampling.

Considering that alias method offers better sampling complexity over ITS, one might opt for the former method for sampling on temporal graphs. However, the dynamically evolving candidate edge set will introduce overwhelming space consumption in the alias method. Particularly, because temporal random walk requires the path to obey the time order, different walkers might need different candidate edge sets even when sampling the same vertex. Using Figure 1 as an example, entering vertex 7 from vertex 9 would lead to the candidate set of {4, 5, 6} while entering vertex 7 from vertex 8 has the candidate set of {0, 1, 2, 3, 4, 5, 6}. In this context, using the alias method alone will require constructing various versions of alias tables. Further, identifying the correct version of the alias table for sampling based on the temporal information of the current arriving edge could also be challenging.

Figure 2 compares *TEA*'s average sampling cost per sampling step (defined as the edges evaluated per step) with two recent works on four temporal datasets. For the two recent works, KnightKing, which relies on rejection sampling, requires the evaluation of 11,071 edges on average per step due to a higher rejection rate. The other approach, GraphWalker [40] adopts a full-scan sampling method that generates all edges of the current candidate edge set to build the alias table on each sampling process and requires an evaluation of 19,046 edges per step. For this method, GraphWalker requires 1 petabyte of preprocessed data for sampling the *twitter* dataset [23]. Compared with these two works, *TEA* only evaluates 5.5 edges on average per step thanks to our hybrid sampling approach (Section 3.2).

Further, there lacks a general-purpose framework with essential algorithmic and system-level optimizations for fast random walks on temporal graphs. This results in poor user productivity and low-performance implementations of these types of algorithms and applications. Additionally, as KnightKing [45] has suggested, it is counter-intuitive for users to implement walker-centric algorithms in popular graph frameworks [29, 54] because programmers could lose the ability to track the walker state updates. Adding another dimension of temporal information to the random walk would further complicate the programmability. Specifically, it would be extremely challenging for the users to manage the dynamically changing sampling space, as well as derive the optimal Monte Carlo sampling method for temporal random walk algorithms.

This paper presents *TEA* which strives to achieve low space consumption, fast sampling speed, and expressive programming interfaces for various temporal random walk applications. At its core, *TEA* provides a novel hybrid sampling method that combines the ITS and alias methods together to drastically reduce space complexity and achieve high sampling speed. This method removes the dependency of edge transition probability calculation on the walker's temporal information and takes advantage of both the ITS and the alias method by averting the expensive searching cost of ITS and the enormous space overhead in the alias method. Here, the sampling space is stored in our novel Persistent Alias Table (PAT) data structure. Furthermore, *TEA* introduces a Hierarchical Persistent Alias Table (HPAT) method, associated with an auxiliary index, to dramatically improve the sampling efficiency, and enable out-of-core sampling for large temporal graphs. Additionally, *TEA* also provides efficient streaming graph processing support. As for programming, *TEA* (written in C++) provides high-level user-friendly APIs and customized function design options to improve user productivity. Finally, our comprehensive performance evaluation reveals that *TEA* can achieve up to $6,158\times$ speedup over GraphWalker and $954\times$ over KnightKing for a diverse range of dynamic random walks on temporal graphs.

The remainder of this paper is organized as follows: Section 2 describes the background. Sections 3 and 4, respectively, present the technical design and system implementation of *TEA*. Section 5 evaluates *TEA*. We discuss the related work in Section 6 and Section 7 concludes.

2 Background

This section discusses temporal graphs and the taxonomy of various Monte Carlo sampling algorithms. The major notations used in this paper have been defined in Table 1.

Table 1. Notation Table.

Notation	Description
D	The maximum vertex degree in graph $\mathbb{G}$
$N(u)$	Vertex $u's$ edge set
$C[k]$	The k-th number of the prefix sum array
$\Gamma_t(u)$	Candidate edge set of vertex u at time t
$\delta((u, v_i, t_i))$	Weight of the edge (u, v_i, t_i)
$P((u, v_i, t_i))$	Edge transition probability of (u, v_i, t_i)
$trunkSize$	The size of each trunk
$\tau_u^{k,i}$	The i-th trunk of vertex u with the length of 2^k
β, p, q	The temporal node2vec parameters
ε	The accepted ratio of rejection sampling

2.1 Temporal Graph

Different from static graphs, edges in temporal graphs include temporal information. Let $\mathbb{G} = (V, E, R)$ be a temporal graph, where V is the vertex set in $\mathbb{G}$, E is the set of edges in $\mathbb{G}$, and R is the temporal information set which is at the size of |E|. Each edge $e \in E$ is defined as a triplet (u, v, t), where $u, v \in V$, t represents the time edge e appears and $t \in R$. We define d_v as the degree of the each vertex v and D is the maximum vertex degree, $D = max\{d_v,\ v \in V\}$. We use the maximum vertex degree for time complexity analysis because, similarly to KnightKing [45], vertices with higher degree numbers will have higher probabilities to be visited in a random walk. A path in a temporal graph is called a temporal path, which starts from a vertex u_1 at time t_1 and arrives at vertex u_n at time t_{n-1}. Thus, this path can be defined by $P = e_1 \cdot e_2 \cdot \ldots e_{n-1}$ where $e_i = (u_i, u_{i+1}, t_i)$, and it must satisfy the time constraint: $t_{i-1} < t_i$ with $i > 1$.

For real-world applications, a temporal graph is represented as an *edge stream*, i.e., a sequence of all edges that come in the order of time when it is created or collected [15, 18, 22, 42, 43, 53]. In this paper, we assume the temporal graph is updated incrementally (see Section 3.5). The e-commerce networks, which add new shopping records to the graphs with respect to temporal information, are a typical example of this dynamic feature. *TEA* adopts the edge stream data representation format for a temporal random walk.

2.2 Monte Carlo Sampling Methods

Various random walk algorithms often follow a similar procedure: a group of walkers, each of which starts from a vertex in a graph (u), selects a neighbor of the current vertex (v_i) from the candidate edge set $N(u)$ (*edge sampling step*), and transits to the selected neighbor. This procedure continues until certain termination conditions are met. Note, the process of selecting a neighbor of the current vertex follows a given probability which is called *edge transition probability* [8, 33, 45]. The edge transition probability for edge (u, v_i) is defined as:

$$P((u, v_i)) = \frac{\delta((u, v_i))}{\sum_{(u, v_j) \in N(u)} \delta((u, v_j))}, \quad (1)$$

where $\delta((u, v_i))$ is the weight of edge (u, v_i).

Sampling edges is the most time-consuming step in random walk [32, 45]. The previous work [45] has reported that the sampling step in a Spark node2vec implementation [9] can take up to 98.8% of the total execution time. Here we briefly introduce three sampling methods that are widely used in random walks, including *inverse transform sampling* [30], *alias method* [27], and *rejection sampling* [39].

Inverse transform sampling (ITS): For each vertex u, ITS uses an array C to store the *Cumulative Distribution Function* by calculating the prefix sum of the weights of $u's$ current edge set, which is defined as $N(u)$. Let us define $N(u) = \{e_1, \ldots, e_i\}$ and assume the weight of each edge e_i as $W(e_i)$, $C[i] = \sum_{j=1}^{i} W(e_j)$. In every sampling process, a random number r is produced in the range of $[0, C[|N(u)|]]$, where $C[|N(u)|]$ is the sum of all the edges' weights in $N(u)$. After this, ITS will find the smallest k that satisfies $C[k-1] <$

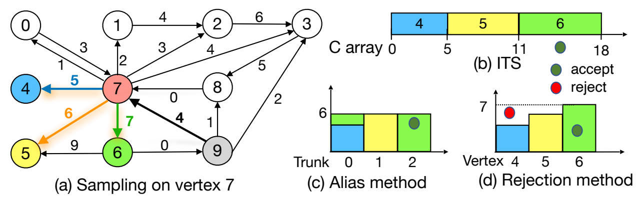


Figure 3. When a walker arrives at 7 from 9, how different Monte Carlo sampling methods, i.e., ITS, Alias Method, and Rejection Sampling, work on the candidate edge set $(7, 4, 5)$, $(7, 5, 6)$, $(7, 6, 7)$, where these edges use the associated temporal information as the sampling weights.

$r \le C[k]$. Thus, the sampled edge will be the kth edge in $N(u)$. This process can be completed via a binary search with the time complexity of $O(log(D))$. Using Figure 3b as an example, when a walker arrives at 7 from 9, it will sample from the edges set $(7, 4, 5)$, $(7, 5, 6)$, $(7, 6, 7)$. The cumulative distribution function C is $C = \{0, 5, 11, 18\}$. Our random number $r = 12$ leads us to select edge $(7, 6, 7)$.

Alias method divides the weight of each edge into several pieces and combines them together to form n trunks. In principle, two conditions have to be satisfied: (1) every trunk can have up to two pieces, and (2) the total weight of each trunk must be the overall average weight. Upon sampling, we first uniformly sample a trunk and subsequently sample a piece in the trunk. Intuitively, the probability of an edge being sampled is proportional to the sum of the weights of its corresponding pieces. The data structures that record these trunks and their contents are called alias tables. The time complexity of generating an alias table and sampling on it is $O(n)$ and $O(1)$, respectively. As shown in Figure 3c, the alias method generates three trunks with an average of 6. Our sampling process selects trunk 2, which only contains one edge. This leads us to select edge $(7, 6, 7)$.

Rejection sampling [39] is recently used to sample the dynamic weight of each edge in a low-dimensional graph [45]. The key advantage of rejection sampling is that when edge weight changes, rejection sampling does not need to regenerate the sampling space because rejection sampling treats each participating edge separately. Once an edge is selected, one only needs to check if this selection is accepted or not. Figure 3d illustrates an example of the rejection sampling on the same vertex 7. Rejection sampling first generates a random number to select a potential edge. In this case, we select vertex 4. Subsequently, one generates another random number to decide whether we should reject or accept this sampled edge. For the second random number, since the range is 0 to the maximum probability across all edges, which is 7 in Figure 3d. Therefore, one could generate a rejected sample like the red dot in Figure 3d. If a sampled edge is rejected, we need to sample again. This process continues until we derive a valid selection, such as vertex 6 in Figure 3d.

2.3 Temporal Random Walk Applications

This section discusses three popular biased temporal random walk applications, which are the most common and complex forms of random walk algorithms. It is worth noting that there also exist unbiased edge weight random walk algorithms. We also want to clarify that despite our *TEA* framework being inspired by biased temporal random walk algorithms, *TEA* can also support unbiased counterparts by assigning uniform weights to all edges.

Candidate edge set: Different from the random walk on a static graph, when a walker arrives at vertex u on a temporal graph, the eligible candidate edges are defined as $\Gamma_t(u)$. Here, $\Gamma_t(u) = \{(u, v_i, t_i) | \ (u, v_i, t_i) \in N(u), \ t_i > t\}$, where t is the time instance associated with the preceding edge that reaches u. Below we describe three popular temporal random walk algorithms.

(I) Linear temporal weight random walk: The edge transition probability for edge $(u, v_i, t_i) \in \Gamma_t(u)$ is defined as:

$$P((u, v_i, t_i)) = \frac{\delta((u, v_i, t_i))}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} \delta((u, v_j, t_j))}, \tag{2}$$

where $\delta((u, v_i, t_i))$ is the weight of edge (u, v_i, t_i). This weight is set as either t_i or $rank((u, v_i, t_i))$. Here, t_i is the time instance associated with this edge (u, v_i, t_i). The *rank()* function is the current edge's timing ranking among all the edges. Since both ways of deriving edge weight are linearly correlated to the temporal information t_i, we consider this variant a linear temporal bias. Recently, CTDNE [31] has implemented this linear weight algorithm to DeepWalk [33].

(II) Exponential temporal weight random walk is another variant of temporal random walk. Using CTDNE [31] as an example, when a walker arrives at a vertex u of time t, the current edge set of u is $N(u)$. The edge weight becomes $\delta((u, v_i, t_i)) = exp(t_i - t)$, which is changing according to the current time instance t. However, since the edge weights of all edges are changing with respect to t, we can cancel out that impact, which is shown in Equation 3. The edge transition probability for each edge $(u, v_i, t_i) \subseteq \Gamma_t(u)$ is:

$$\begin{aligned} P((u, v_i, t_i)) &= \frac{\delta((u, v_i, t_i))}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} \delta((u, v_j, t_j))} \\ &= \frac{exp(t_i - t)}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} exp(t_j - t)} \\ &= \frac{exp(t_i)}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} exp(t_j)}. \end{aligned} \tag{3}$$

The exponential temporal weight random walk is widely used in temporal graphs to capture time instances such as CAW [41] and EHNA [17]. The exponential function here is similar to the exponentially decaying probability of consecutive contacts, which has been observed in the spread of computer viruses and worms [14].

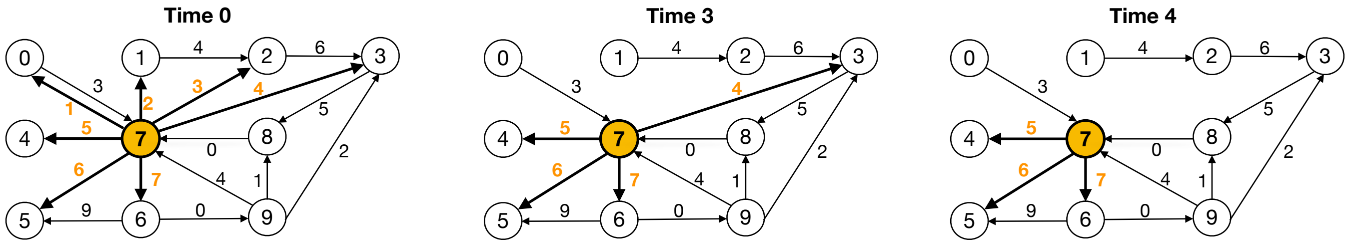


Figure 4. Evolution of temporal graph i.e., candidate edge set and candidate edge weight with time for vertex 7.

(III) Temporal node2vec [17] extends CTDNE according to the definition of node2vec [8]. The probability distribution depends on not only the time instance of the preceding vertex but also the distance between the preceding vertex and the candidate vertex. When a walker arrives at a vertex u of time t with w as the preceding vertex of u in the random walk, the edge transition probability for edge $(u, v_i, t_i) \subseteq \Gamma_t(u)$ can be expressed as :

$$P((u, v_i, t_i)) = \beta_{(u,v_i)} \cdot \frac{\delta((u, v_i, t_i))}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} \delta((u, v_j, t_j))}, \quad (4)$$

where $\beta_{(u,v_i)}$ is $\frac{1}{p}$ if $d_{(w,v_i)} = 0$, 1 if $d_{(w,v_i)} = 1$, and $\frac{1}{q}$ if $d_{(w,v_i)} = 2$.

The definitions of $\beta_{(u,v_i)}$, $d_{(w,v_i)}$, p, and q are the same as their definitions in node2vec [8] on static graphs. Particularly, p and q control the random walk to walk like either Breadth- or Depth- First Search algorithms. With $d_{(w,v_i)} = 0$, u will travel back to w. With $d_{(w,v_i)} = 1$, v_i is one-hop away from u and w. With $d_{(w,v_i)} = 2$, v_i is two-hops away from w. $\beta_{(u,v_i)}$ is combined with $exp(t_i - t)$ to get more time-sensitive information from the temporal paths.

3 *TEA*: A Temporal Graph Random Walk Engine

3.1 Observation and Overview

Observation. In this paper, we observe that, for a temporal random walk, using any Monte Carlo sampling algorithm alone will suffer from overwhelming performance challenges. Below we offer our key observations:

First, the rejection sampling method faces an extremely large number of trials when applied to temporal graphs. For example, when random walks arrive at vertex 7 from vertex 8 in Figure 1, the candidate set for vertex 7 will be $\{0, 1, \ldots, 6\}$ and the weight distribution of the exponential temporal weight random walk will be $\{exp(1), exp(2), \ldots, exp(7)\}$. The expected trials can be as large as $\frac{7*exp(7)}{\sum_{j=1}^{7} exp(j)}$. Figure 2 also confirms a higher average sampling cost for rejection sampling with KnightKing.

Second, ITS sampling method always experiences $O(log(D))$ time complexity, which is nontrivial. It is important to note that for different time ranges of interest, ITS will not recompute the sampling space to reduce the searching space because reconstructing the sampling space is often more time-consuming than directly sampling on the largest time range. Therefore the sampling complexity remains to be $O(log(D))$ for ITS.

Third, for the alias method, one would have to build one version of the alias table for each unique time step t of each vertex to take advantage of precomputing the transition probability for fast sampling. For each vertex v_i, the alias method stores the alias tables of all possible candidate sets of v_i. The alias table of each candidate set will take $O(D_{v_i})$ space with D_{v_i} as the degree number of v_i. The space complexity is $O(D_{v_i}^2)$. Therefore, the alias method would require around $\sum_{v_i \in V} D_{v_i}^2$ space to store all the alias tables. Such an enormous space consumption will make the storing and indexing of the alias table (i.e., deciding which alias table to search against) prohibitively expensive. For example, for vertex 7 in Figure 4, the candidate set for vertex 7 will be different if the arriving time is 0, 3, or 4. Three candidate edge sets will lead to three different alias tables for the same vertex 7.

Overview. To combat the high sampling cost faced by rejection sampling, nontrivial sampling time complexity by ITS, and the enormous space requirement faced by alias table method-based temporal random walk, *TEA* introduces a hybrid approach that combines both ITS sampling and alias method. Particularly, the edge sets are partitioned into a collection of smaller static partitions, each of which is a trunk of an alias table. During sampling, ITS sampling is used to select the trunk of interest. Subsequently, inside each trunk, we will resort to the alias method for sampling. Furthermore, *TEA* proposes an HPAT design for in-memory sampling and auxiliary indexing for fast identifying the alias table of interest. Finally, considering temporal graphs could come in as a streaming format, *TEA* also introduces streaming graph support.

3.2 Persistent Alias Table (PAT)

Although the candidate edge sets could change dynamically according to the temporal information in the temporal graph, each could be a combination of several smaller subsets of edges. In our PAT method, we partition the entire edge set into several smaller subsets. Here, each subset of edges also referred to as a trunk, remains unchanged. For a candidate edge set that contains multiple trunks, we resort to the ITS sampling method to select the trunk of interest. Finally, since the trunk remains unchanged, one can use the alias method to perform sampling in that trunk.

PAT encompasses new data structures and sampling algorithms to allow efficient sampling. For the new data structure, we construct alias tables for all the trunks and the prefix-sum array at the granularity of the trunk. Our construction process works as follows. We first partition the entire neighbor list into a collection of trunks, each containing an equal number of edges. As shown in Figure 5, we partition vertex 7's neighbor list into four trunks, {6, 5}, {4, 3}, {2, 1}, and {0} in a decreasing time order. Afterward, we build an alias table for each trunk. In the meantime, we will construct the prefix-sum array for these four trunks. Here, we assume the temporal weight of each edge is defined by the linear temporal weight rule discussed in Section 2.3, which is shown as the temporal weight in Figure 5. In this case, we can build the alias table for each trunk with the mean as 6.5, 4.5, 2.5, and 1, respectively. The subsequent prefix-sum array of the trunks are {0, 13, 22, 27, 28} as shown in the bottom right of Figure 5. Note that we name this data structure *persistent alias table (PAT)* referencing the concept of persistent segment tree [34].

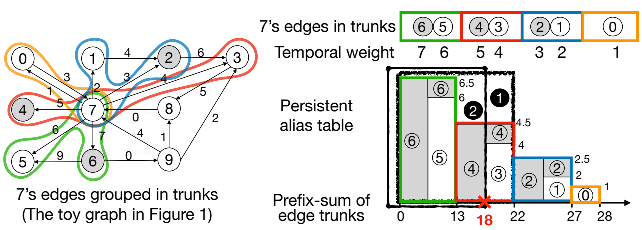


Figure 5. Persistent alias table (PAT) for vertex 7 of Figure 1 with the edges arranged by decreasing time. Under this new construction, ITS is first used to select a particular trunk, then the alias method is applied to perform sampling inside the selected trunk.

Atop our PAT data structure design, our algorithm would need to cope with two types of sampling cases hinging upon whether all the edges in the trunk selected from ITS are complete or not. The first sampling case, when all the edges in the selected trunk are complete, is straightforward. Simply, alias method can then be used directly to sample the neighbor of interest within the trunk. This case is demonstrated as ❶ in Figure 5. Assuming the incoming edge is $(0, 7, 3)$, the candidate set is $\{6, 5, 4, 3\}$. The selected trunk is complete with a prefix-sum range of 0 to 22. Hence, alias method can be directly used to sample a neighbor of interest.

The second case needs to deal with sampling within an incomplete trunk. If the selected trunk by ITS is incomplete, alias method cannot be used as alias method can only be performed on a complete trunk. In that case, ITS is used to sample within the trunk by rebuilding the prefix-sum of that trunk. Taking the ❷ in Figure 5 as an example, assuming the incoming edge is $(9, 7, 4)$, the candidate edge set will be $\{6, 5, 4\}$. This candidate set occupies the whole trunk $\{6, 5\}$ and a part of the trunk $\{4, 3\}$ (edge 3 is not included). In this case, PAT builds the prefix-sum array inside the incomplete trunk, e.g., the prefix-sum array of the edge set $\{4\}$, and then performs the ITS on it for sampling.

Intuitively, our PAT method alleviates the drawbacks of both the alias table and ITS methods. First, compared to the alias method design, for each vertex u, we reduce the space consumption from $O(D^2)$ to $O(D)$, where D is the degree of vertex u. Note that our method only takes $O(D)$ space because our alias table trunks take $O(D)$ space and prefix-sum of trunks only takes $O(\frac{D}{trunkSize})$ space. Second, compared to the ITS method design, our PAT method can reduce the searching time complexity from $O(\log D)$ to $O(\log \frac{D}{trunkSize})$.

The *trunkSize* selection strategies differ under various execution modes. When the memory capacity is sufficient (full-in-memory execution), the *trunkSize* should be as large as possible while satisfying that the time complexity of ITS on the prefix-sum of trunks (i.e., $O(\log \frac{D}{trunkSize})$) is not smaller than the time complexity of ITS inside each trunk (i.e., $O(\log trunkSize)$). Hence, *trunkSize* should not be larger than $\sqrt{D}$. In this case, we can choose *trunkSize* as $\lfloor \sqrt{D} \rfloor$ for each vertex. When the memory is insufficient, we will run PAT under the out-of-core execution mode. To reduce the disk I/O, we choose the *trunkSize* as small as possible while satisfying that we have enough memory space to store the prefix-sum array of all trunks whose size is $\frac{|E|}{trunkSize}$. For example, we can choose *trunkSize* as 10 on the twitter dataset under 16 GB memory limitation.

3.3 Hierarchical PAT (HPAT)

Although our PAT design can dramatically reduce space consumption, it still has the $O(\log \frac{D}{trunkSize})$ time complexity. Thus, we propose a *hierarchical persistent alias table design (HPAT)* method to trade slightly more memory space for lower sampling complexity.

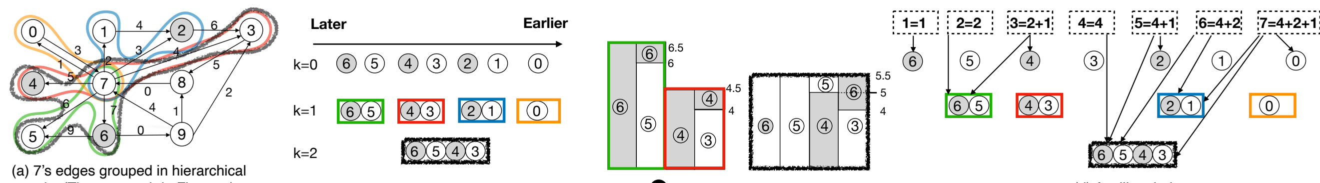
Equations 5, 6 and 7 formally define how we construct our HPAT for each vertex u with edge set as $\{e_1, \ldots, e_n\}$ in a hierarchical manner.

$$\tau_u = \{\tau_u^0, \ldots, \tau_u^k, \ldots, \tau_u^K\}, \quad 0 \le k \le K = \lfloor \log_2(|N(u)|) \rfloor. \tag{5}$$

$$\tau_u^k = \{\tau_u^{k,0}, \ldots, \tau_u^{k,i}, \ldots, \tau_u^{k,I}\}, \quad 0 \le i \le I = \lfloor \frac{|N(u)|}{2^k} \rfloor - 1. \tag{6}$$

$$\tau_u^{k,i} = \{e_{i*2^k+1}, \ldots, e_{(i+1)*2^k}\}. \tag{7}$$

In Equation 5, each τ_u^k is a relatively bigger trunk. Subsequently, in Equation 6, we partition each bigger trunk τ_u^k



(a) 7's edges grouped in hierarchical trunks (The toy graph in Figure 1). (b) Sequentially group 2^k edges into a trunk. (c) Case ❶ without (left) vs. with (right) HPAT. (d) Auxiliary index.

Figure 6. Hierarchical persistent alias table design (HPAT) for vertex 7 of Figure 1: (a) 7's edge grouped in hierarchical trunks (the toy graph in Figure 1), (b) sequentially group each 2^k edges into a trunk, (c) case ❶ without (left) vs. with (right) HPAT, and (d) the auxiliary index for HPAT.

from Equation 5 into smaller trunks, i.e., $\tau_u^{k,i}$. Further, as shown in Equation 7, each trunk is represented by an edge set of $\tau_u^{k,i}$, which represents the i-th trunk of vertex u with the length of 2^k. For each trunk (bigger or smaller), we build an alias table for subsequent sampling.

When sampling occurs, since each candidate edge set $\{e_1, \ldots, e_i\}$ must be the prefix of the current vertex's edge set $\{e_1, \ldots, e_n\}$ with decreasing time, the candidate edge set can be divided into a number of trunks via binary decomposition. Then, *TEA* first samples these trunks using *ITS* to reduce the overall space overhead. After this, the alias table of the sampled trunk is used to locally sample an edge (i.e., *alias method* is applied here to enable fast sampling).

Still using vertex 7 of Figure 1 as an example, the candidate set will be $\{6, \ldots, 0\}$. Our trunk sets are shown in Figure 6b, τ_u can be represented as a set of τ_u^0=$\{\{6\}, \ldots, \{0\}\}$, τ_u^1= $\{\{6, 5\}, \{4, 3\}, \{2, 1\}\}$, τ_u^2=$\{\{6, 5, 4, 3\}\}\}$. Upon sampling, the candidate set is divided into three trunks ($7 = 4 + 2 + 1$): Trunk_set=$\{g_1, g_2, g_3\}$, where $g_1 = \{6, 5, 4, 3\} = \tau_u^{2,0}$, $g_2 = \{2, 1\} = \tau_u^{1,2}$, and $g_3 = \{0\} = \tau_u^{0,6}$. Then the sampled probability of each trunk can be calculated by using the *ITS* array C: $P(g_1) = (0, \frac{C[4]}{C[7]}]$, $P(g_2) = (\frac{C[4]}{C[7]}, \frac{C[6]}{C[7]}]$, $P(g_3) = (\frac{C[6]}{C[7]}, 1]$. After all available trunks are sampled by *ITS*, sampling based on the local alias method begins: an edge in the sampled trunk is sampled by the alias table in the trunk.

In this design, the time complexity of PAT is further reduced to $O(log(log(D)))$ because each candidate edge set will cover up to $log(D)$ trunks. After that, the local processing using the alias method within each sampled trunk only takes $O(1)$ time complexity. In terms of space consumption, only the alias tables of the subsets of $\tau_u^{k,i}$ need to be preprocessed, resulting in the space overhead of τ_u^k as D and the overall space overhead of τ_u as $O(Dlog(D))$ with D as the degree number. This is still much lower than simply applying the alias method, which costs $O(D^2)$ for random walking on temporal graphs. Although it has a higher space overhead than *ITS*, which only needs D space, our sampling methods have a faster sampling speed.

We also added two *ad hoc* optimizations. First, if the temporal information of certain neighbors is earlier than all the incoming edges, we can simply discard them. Second, if the out-degree of a vertex is relatively low, we can simply build alias tables for its specific out edges.

3.4 Auxiliary Index

This section introduces the concept of the auxiliary index to reduce the time complexity for finding the trunks of interest for each candidate edge set from $O(log(D))$ to $O(1)$. During sampling, HPAT designs need to find the trunks containing the candidate edges, i.e., $\Gamma_t(u)$, which would lead to $O(log(D))$ time complexity. Below we detail the complexity analysis. When a walker arrives at vertex u of time t, *we need to find the minimum number of trunks at sizes of* 2^i *that can construct* $\Gamma_t(u)$. This process needs $log(|\Gamma_t(u)|)$ operations for decomposing $\Gamma_t(u)|$ into appropriately sized trunks, and $log(D)$ operations to find those trunks of interest. Using vertex 7 in Figure 1 as an example, when a walker arrives at vertex 7 from 0, the candidate edge set is $\Gamma_{t=3}(u) = \{6, 5, 4, 3\}$. For HPAT, the required trunk will be $\{6, 5, 4, 3\}$. In another situation, if vertex 7 is arrived from vertex 9, the $\Gamma_{t=4}(u)$ would be $\{6, 5, 4\}$. In this case, the trunks of interest in the hierarchical persistent alias method would be $\{6, 5\}$ and $\{4\}$.

Our auxiliary index enables *TEA* to rapidly identify the trunks of interest. Figure 6d shows how to build an auxiliary index for HPAT of vertex 7. Assuming the HPAT requires 7 neighbors from vertex 7, i.e., the top right dotted box in Figure 6d, these neighbors will fall into three trunks: {6, 5, 4, 3}, {2, 1} and {0}. Since the HPAT arranges all the alias tables into a complete binary search tree, with the indices $4 + 2 + 1$, we can locate the trunks of interest as follows. First, value 4 indicates we should fetch the only size =4 trunk in the top-level, i.e., {6, 5, 4, 3}. Second, the value of 2 indicates our second trunk of interest lies in the second level of the binary search tree. Then the position of the trunk would start from 4, which is the sum of the size of the prior trunk. Finally, value 1 indicates that this trunk resides in the third level (where the size of all the trunks is 1). Further, the position of the current trunk would be the sum of the sizes of fetched trunks, that is, 4+2=6. Therefore, we obtain the last trunk. This design reduces the time complexity of trunk finding from $O(log(D))$ to $O(1)$.

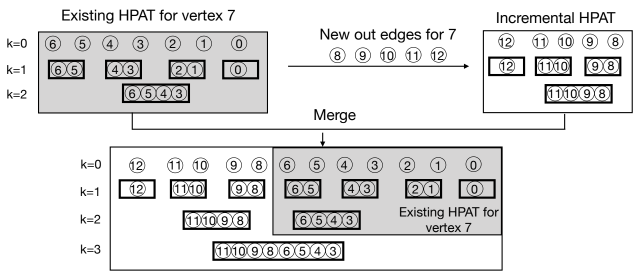


Figure 7. Incremental HPAT updating for vertex 7 of Figure 1 with five new streaming edges from 7 to {8, 9, 10, 11, 12}, respectively.

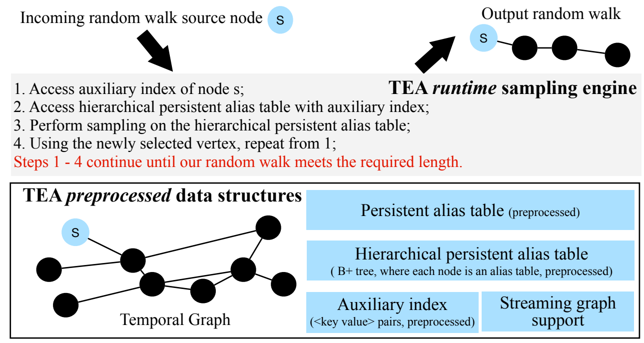


Figure 8. Overall workflow of *TEA*.

3.5 Streaming Graph Support

For the streaming graph setting, we assume the updates include the addition of new edges and vertices, and the updates are done in batches for the sake of efficiency, which is similar to the state of the art streaming graph systems [2, 20, 36]. For each new batch, we need to update the PAT and HPAT indices. Fortunately, both our PAT and HPAT are built by the timing order of the edges, that is, we arrange the edges by decreasing timing order. For the batch of new incoming edges, since their timing information is always larger than the existing edges, we can simply append these new edges to PAT and HPAT. That leads to our incremental update design for PAT and HPAT. Since PAT is a special case of HPAT, we describe the incremental update support for HPAT below.

The key to our incremental update design is that - we keep the old HPAT index intact and create new trunks for the incoming edges. Chances are that the newly added trunks together with the old ones could lead to the growth of the hierarchy in HPAT. We hence increase the hierarchy by combining the existing and new trunks. Figure 7 exemplifies this design for the same sample graph in Figure 6b. Assuming the new incoming batch contains edges from vertex 7 of Figure 1 to {8, 9, 10, 11, 12}, where these incoming edges are sorted by increasing time order. We perform the incremental update to HPAT in two steps. First, we build an incremental HPAT for these new arrivals (top right of Figure 7). Second, when merging our incremental HPAT and the existing HPATs, we might also need to generate a higher hierarchy for our HPAT like {3, 4, 5, 6, 8, 9, 10, 11} at the bottom of Figure 7. Because we create new HPATs with a higher hierarchy, even under the out-of-core mode, the created HPATs will be stored sequentially following current HPATs.

4 System Implementation

4.1 *TEA*: Putting All Things Together

Figure 8 presents the workflow of random walk atop *TEA*. Particularly, for each random walk step, *TEA* uses the active vertex to access the corresponding auxiliary index. And the resultant auxiliary index is used to index the respective HPAT. Finally, we perform sampling on the HPAT to derive the sampled vertex. This process continues until the convergence (i.e., arriving at the random walk length). *TEA* outputs the sampled path at the end.

Table 2. *TEA* API specifications.

Name	Description
Dynamic_weight()	Dynamic weight definition interface
Dynamic_parameter()	Dynamic parameter
Edges_interval()	Extract the temporal subgraph

Algorithm 1 Temporal node2vec in temporal-centric API.

```
Dynamic_weight (Time time)
    return exp(time)

Dynamic_parameter (Vertex u, Vertex v)
    if(u==v)
        return 1/p
    else if(u.ISNEIGHBOR(v))
        return 1
    else
        return 1/q

Edges_interval(Edges_set E, Time start_time, Time end_time)
    begin=E.find(start_time)
    end=E.find(end_time)
    return {E[begin] , ..., E[end]}
```

***TEA* Framework and temporal-centric API:** We provide a *temporal-centric* framework for the end-users to express temporal random walk algorithms with ease. We extend the walker-centric concept on KnightKing to our **temporal-centric** one which lets users think from the "time" perspective: The time instance affects the core of random walk, that is, probability distribution. This framework mainly consists of two user involvements, i.e., parameters and subgraph selection which are listed in Table 2. Parameters (e.g., **Dynamic_weight** and **Dynamic_parameter**) allow users to

Algorithm 2 Temporal-centric API in TEA.

```
Sampling (Random R, Vertex u, Time t)
  Candidate edge set L = Γ_t(u)
  Trunks set L' = Auxiliary_Index (L)
  Sampled trunk index (k,i) = Sampling R on L' by ITS
  return Edge = Sampling R on the trunk τ_u^{k,i} by alias method

Preprocess (Edges_set E, Dynamic_weight())
  Dynamic_weight() defines the weight for each edge in E
  Generate HPAT for E

Main (Len, start_time, end_time)
  E' = Edges_interval(E, start_time, end_time)
  Preprocess(E', Dynamic_weight( ))
  while (Len>0)
    for (each random walk S)
      (u, t) = S.current_vertex
      (u_p, t_p) = S.previous_vertex
      while (True)
        (R, R') = random()
        (u, v, t') = Sampling(R, u, t)
        if (R' ≤ Dynamic_parameter(u_p, v))
          break
      S.previous_vertex = (u, t)
      S.current_vertex = (v, t')
    Len = Len-1
```

customize the bias according to different applications. Subgraph selection provides more expressiveness to users, i.e., (**Edges_interval**). **Edges_interval** derives the subgraphs of interest for *TEA* to perform random walk on. All these APIs center around temporal information.

Algorithm 1 shows the usage of our APIs for temporal node2vec. **Dynamic_weight** is defined as $exp(time)$ as in the state-of-the-art [31]. **Dynamic_parameter** defines the parameter $\beta_{(u,v_i)}$ in Equation 4 of temporal node2vec. **Edges_interval** is provided for users to generate a subgraph (snapshot) for random walk according to different applications.

Algorithm 2 shows how user APIs interact with the *TEA* framework. Particularly, it first uses **Edges_interval** to get the subgraph for each query. Then, *TEA* uses **Preprocess** function to generate the alias tables and auxiliary index. During random walk computation, **Sampling** function uses HPAT to encode **Dynamic_weight** and the random number R. Then *TEA* uses the rejection sampling to check whether this sampling trial is in the "Accepted" area, i.e., whether the random number R' is not larger than the dynamic parameters provided by **Dynamic_parameter**. For random walks without dynamic parameters, we simply return "Accepted" during each rejection sampling process. Finally, it updates random walks by newly sampled edges.

Out-of-core support: While HPAT is faster, it requires more space. Our out-of-memory case focuses on space-saving when HPAT cannot fit in memory. First, we resort to PAT which is smaller than HPAT for sampling. During sampling, the prefix-sum array of each edge's trunk is cached in memory (Section 3.2) and we use it to sample the trunk of interest, which will be loaded into memory for alias table-based sampling. Our workflow of out-of-core execution is similar to GraphWalker [40] except that we use PAT for sampling. As shown in Section 3, if the sampled trunk is completed (Section 3.2), we load the alias table of the current trunk into memory, otherwise, we load the prefix-sum array. Second, each to-be-loaded data will use the prior LOADED DATA RE-ENTRY [1] to minimize the disk I/O. The updating process uses multiple threads to update walkers asynchronously. Third, *TEA* stores the completed random walks the same as GraphWalker [40], that is, we flush the completed ones to disk when the number of them reaches 1,024.

4.2 Parallel *TEA* Data Structure Construction

TEA mainly requires the following three components: (1) search the candidate edge set of each edge (2) construct the PAT/HPAT for each vertex, and (3) generate the auxiliary index for the HPATs. Fortunately, we can perform all these steps in parallel. In the following, we analyze the above three processes in detail.

Searching candidate edge sets: On sampling, when the current random walk arrives at edge (u, v, t), it needs to find the candidate edge set $\Gamma_t(v)$, which are the out-edges that are later than the time t of edge (u, v, t). We search the candidate edge sets for all in-edges in parallel in two steps. First, we sort the out-edges of the same source vertex in time decreasing order by the radix sort with $O(|E|)$ time complexity. Second, we perform a binary search on the sorted out-edge list to determine the candidate edge set for each in-edge with $O(|E|log(D))$. However, this step can be conducted in parallel because the candidate edge set for each in-edge is independent.

PAT/HPAT construction: *TEA* needs to construct alias tables for both PAT and HPAT. For brevity, we mainly discuss the HPAT construction because the PAT can be constructed similarly. If different threads are assigned to construct different alias tables, threads may compete for the same memory position. To provide a lock-free parallel construction process, we calculate the position of each alias table ($\tau_u^{k,i}$) in memory before construction. As the length of each alias table is fixed (i.e., 2^k), the position can be calculated before constructing the alias table. With the derived position, we can assign a thread to construct each alias table and store the resultant alias table in the designated memory position without contentions.

Auxiliary index generation: The auxiliary index is constructed on each candidate edge set $\Gamma_t(u)$ for HPAT sampling. As the size of $\Gamma_t(u)$ is up to the degree size of u, it can store the binary decomposition of each degree size from 1 to D, where D is the maximum degree of the whole graph. Therefore, the auxiliary index construction takes $\sum_{D'=1}^{D} log(D')$

time. For most traditional graphs, the maximum degree D is up to millions, which leads to the acceptable auxiliary index construction time. Additionally, the binary decomposition of different candidate edge sets is independent. Therefore, the auxiliary index construction can be parallelized embarrassingly.

4.3 Complexity Analysis

This part compares the time complexity of *TEA* with state-of-the-art works GraphWalker [40] and KnightKing [45] for linear temporal weight random walk, exponential temporal weight random walk and temporal node2vec algorithms. *TEA* uses HPAT for all random walk algorithms. For linear temporal weight random walk, both GraphWalker and KnightKing use *ITS*. For exponential temporal weight random walk, GraphWalker uses the sequential pass to calculate the probability distribution and then samples the selected edge, while KnightKing relies upon the rejection sampling method. When dealing with the temporal node2vec, for dynamic weight component, GraphWalker uses the full-scan sampling method (Section 1), and KnightKing exploits rejection sampling. For the dynamic parameter component, both GraphWalker and KnightKing use rejection sampling.

For the linear temporal weight random walk, both GraphWalker and KnightKing have $O(log(D))$ time complexity while *TEA* only has $O(log(log(D)))$ time complexity thanks to our novel hybrid sampling approach on the HPAT data structure. For the exponential temporal weight random walk, the time complexity of GraphWalker is $O(D)$ for the sequential pass, and KnightKing is $O(\frac{1}{\varepsilon})$ where ε is the accepted ratio of rejection sampling. The accepted ratio can be defined as $\varepsilon = \frac{\delta(e_1)+\delta(e_2)+\ldots+\delta(e_D)}{D*\delta(e_D)}$ where $\delta(e_i)$ is the weight of edge e_i and the maximum weight of these edges is $\delta(e_D)$. For example, as discussed in Section 3.1, the accepted ratio of rejection sampling of vertex 7 is $\varepsilon = \frac{\sum_{j=1}^{D} exp(j)}{D*exp(D)}$. Because the accepted ratio is always larger than $\frac{1}{D}$, the time complexity of KnightKing is slightly smaller than $O(D)$. For *TEA*, the complexity is always $O(log(log(D)))$.

The only difference between temporal node2vec and exponential temporal weight random walk is that the temporal node2vec has the dynamic parameter β which is defined in Equation 4. The dynamic weight for both algorithms is the same as defined in Equation 3. For the dynamic parameter β, both KnightKing and GraphWalker use the rejection sampling as β is requiring a small and constant expected number of trails. Therefore, for both KnightKing and GraphWalker, the time complexity of temporal node2vec is the same as the exponential temporal weight random walk.

4.4 Discussion and Limitations

We anticipate that *TEA* could offer two benefits to the broader community. First, the training of temporal graph neural networks on large graphs, such as recent work [16], could benefit from *TEA*. Particularly, sampling is one of the most expensive steps for training a GNN on large graphs. Since *TEA* could accelerate sampling by orders of magnitude, the impacts on GNN training for temporal graphs would be enormous. Second, random walks and sampling on static graphs could also benefit from our idea of combining various Monte Carlo sampling methods together. For instance, C-SAW [32], which scales the random number to perform repeated sampling, could benefit from our hybrid approach.

Although *TEA* proposes efficient solutions for temporal graph random walk, there are still some limitations. First, *TEA* can not support distributed random walk and sampling. One possible solution could be replacing the rejection sampling of KnightKing [45] by our PAT or HPAT in order to support distributed execution. Second, *TEA* can only support streaming graphs. Other cases such as deleting or changing vertices or edges are not supported. We plan to add support for these features to *TEA* in the future.

5 Evaluation

5.1 Experimental Setup

Environment: *TEA* is evaluated against GraphWalker and KnightKing both of which are state-of-the-art general random walk engines. We use two setups i.e., a single machine and a distributed machine for evaluation. While *TEA* is evaluated in both single and distributed machine, GraphWalker is evaluated on the single machine and KnightKing is evaluated on a distributed machine. For single machine execution, evaluations are performed on a machine with two Intel(R) Xeon(R) CPU E5- 2640 v2 @ 2.00GHz (each has 8-cores), 94GB DRAM (20MB L3 Cache) and an 1TB SATA SSD (650MB/s for sequential read). For distributed machine, we use an 8-node cluster (each node is the same as our single machine configuration) with 40Gbps IB interconnection.

Table 3. Datasets used for evaluation (where k = 10^3).

Dataset	|V|	|E|	Degree Mean	Max Degree
growth	1,870k	39,953k	42.714	226,577
edit	21,504k	266,769k	21.069	3,270,682
delicious	33,777k	301,183k	66.752	4,358,622
twitter	41,652k	1468,365k	74.678	3,691,240

Benchmarks: Table 3 lists the details of datasets in this section. We choose four widely used datasets from Koblenz Large Network Collection [23] for evaluation, all of which are temporal graphs in the standard format, i.e., edge streams. For each graph, $|V|$ denotes the number of vertices; $|E|$ denotes the number of edges; *Degree Mean* gives the averaged

Table 4. The runtime performance (in seconds) and speedup of *TEA* over state-of-the-art methods GraphWalker and KnightKing.

Datasets	Linear weight random walk			Exponential weight random walk			Temporal node2vec		
	GraphWalker	KnightKing	*TEA*	GraphWalker	KnightKing	*TEA*	GraphWalker	KnightKing	*TEA*
growth	14.97 (26.4x)	2.46 (4.3x)	0.56	39.71 (13.6x)	4.82 (1.6x)	2.93	52.18 (14.8x)	7.03 (2.0x)	3.52
edit	161.12 (30.9x)	25.8 (4.9x)	5.21	27961.48 (860.1x)	2583.94 (79.5x)	32.51	71907.56 (1536.1x)	10388.17 (221.9x)	46.81
delicious	248.36 (31.1x)	40.60 (5.1x)	7.98	46479.26 (1196.6x)	5044.26 (129.9x)	38.84	119724.11 (2001.5x)	29627.98 (495.3x)	59.82
twitter	479.84 (39.4x)	73.26 (6.0x)	12.16	224421.26 (3140.0x)	37968.30 (531.3x)	71.47	572274.20 (6158.0x)	88677.35 (954.2x)	92.93

number of edges connected to each node; *Max Degree* is the maximum degree among all nodes in the graph. These datasets, especially the large ones, are representative power-law graphs. As discussed previously, since evolving graphs can be transformed into temporal graph representation [25] for random walking, we exclude the evaluation for them.

Walkers parameters: For traditional random walk algorithms, we have to set the number of walks starting from each vertex (i.e., R), which gives the total number of walks as $R * |V|$. For fairness, we set the number R as 1 and the maximum length $L = 80$, which is the same as traditional random walk engines such as KnightKing [45]. For the dynamic parameters p and q of the temporal node2vec, we set $p = 0.5$, $q = 2$ which is widely used in random walk engines [17, 45].

Baselines: For the all-in-memory mode, we compare *TEA* with both GraphWalker [40] and KnightKing [45]. We directly use open source codes of GraphWalker and KnightKing. KnightKing uses 8 nodes distributed setting which has better performance than only on a single node [45]. Note that both GraphWalker and KnightKing use binary search to search candidate edge sets on sampling, while *TEA* does not. For the external-memory mode, we only compare *TEA* to GraphWalker because GraphWalker can support external-memory well while KnightKing cannot.

5.2 *TEA* vs. State-of-the-art Systems

Table 4 presents the overall performance of GraphWalker, KnightKing, and our *TEA*. For fairness, we include the preprocessing time of *TEA* in the total random walk time.

Linear temporal weight random walk: We evaluate GraphWalker, KnightKing, and *TEA* on linear temporal weight random walk under different datasets to demonstrate the effectiveness of *TEA*. Overall, *TEA* is $26.4 \sim 39.4\times$ faster than GraphWalker. Because KnightKing uses eight machines, the speedup of *TEA* over KnightKing is lower than that of GraphWalker. However, we still achieve a $4.3 \sim 6.0\times$ speedup over KnightKing. Because our main sources of performance improvement come from the optimization of the candidate edge sets searching in preprocessing and the sampling process, both of which are associated with the graph degree, we observe that the graph dataset that exhibits the maximum speedup for *TEA* over GraphWalker to be the same for KnightKing too. Similarly for the minimum speedup case.

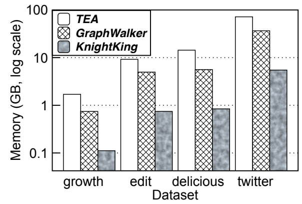


Figure 9. Memory Usage.

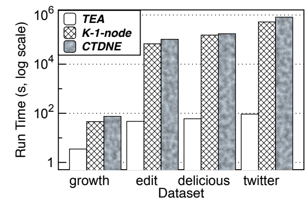


Figure 10. *TEA* VS Others.

Exponential temporal weight random walk: Because exponential temporal weight random walk has to deal with dynamic edge weights, GraphWalker needs to rebuild the transition probability on demand. This process has to fully scan all the neighbors. Together with sampling, GraphWalker takes about 62.3 hours on the largest dataset *twitter*. In contrast, *TEA* finishes the entire sampling in 1.2 minutes. Overall, *TEA* can achieve up to $3,140\times$ speedup across all settings. Even for the smallest dataset, we still observe a $13.6\times$ speedup. Although the exponential function makes the probability distribution highly skewed which will result in a large number of trials, KnightKing still outperforms GraphWalker because the number of trials is always less than the degree of nodes D, which determines the overhead of each sampling in GraphWalker. Further, because KnightKing is a distributed system that runs on an 8-node cluster, KnightKing has several times speedup over GraphWalker. But overall, our *TEA* achieves up to $531\times$ speedup over KnightKing. As KnightKing uses rejection sampling, it requires a large number of trials which follows the discussion of Section 4.3.

Temporal node2vec: Different from the exponential random walk, temporal node2vec further adds the dynamic parameter β (Equation 4) into random walk generation and the time complexity of temporal node2vec is close to β as shown in Section 4.3. For the dynamic parameter β, all systems use rejection sampling. As current works always choose $p = 0.5$ and $q = 2$ for β, the expected trial number is not large [45]. For each trail, temporal node2vec runs a exponential random walk. As large degree vertices tend to have higher trail numbers and *TEA* can achieve better performance on the large degree numbers, *TEA* has better improvement on temporal node2vec than the exponential random walk. Particularly, the speedup of *TEA* is $14.84 \sim 6,158\times$ over GraphWalker and up to $954\times$ over KnightKing.

Memory comparison: Figure 9 illustrates the memory usage of *TEA*, GraphWalker, and KnightKing on different datasets. *TEA* uses HPAT data structure under the full memory mode. *TEA* takes up to 78.06 GB on twitter, while using 2 GB memory on growth. The HPAT index takes the most space, i.e. 82.5% ~ 91.2%, of the total memory usage. Compared with state-of-the-art systems, GraphWalker takes 36.48 GB on twitter, while KnightKing takes a maximum of 6.91 GB per node under 8-node distributed execution. When KnightKing is executed in a single node, it takes 45 GB on twitter. While HPAT takes a slightly larger space than the state-of-the-art, it is astonishingly 6,158× and 954× faster than GraphWalker and KnightKing on this twitter graph, respectively.

Compare with other engines: Figure 10 compares *TEA* with other engines, including KnightKing under the single node execution (K-1-node) and CTDNE [31] with the temporal node2vec random walk method. The general trend is that our *TEA* outperforms both K-1-node and CTDNE tremendously. Particularly, *TEA* can achieve 5,627× speedup over K-1-node. Compared to CTDNE, *TEA* can achieve up to 8,816× speedup because CTDNE provides a temporal graph random walk-based neural network model rather than an efficient random walk system with system-level optimizations.

Parameters Sensitivity: Random walk parameters affect the overall performance. For R and L, they directly affect the performance. The runtime of $R = 2$ is 1.91 ~ 2.14× longer than R=1 under different L with range from 10 to 80. The runtime of $L = 80$ takes nearly 4.7 ~ 5.9× longer runtime than L=10 under different R with range from 1 to 3.

Applications scope: We also notice that some other popular static graph random walk based algorithms, such as SimRank [19], meta-path [6], and Personalized PageRank [11, 46], do not have existing variations on temporal graphs. Nevertheless, if practitioners would like to implement these applications atop temporal graphs, the goal can be conveniently achieved by deploying them atop *TEA*, which comes with algorithm-level and system-level optimizations together with the general framework provided by *TEA*.

5.3 Piecewise Breakdown

In this section, we study the impacts of our two major optimizations, HPAT sampling optimization, and auxiliary index optimization (Section 3.4). We choose the temporal node2vec as the example application and perform this study under the in-memory environment. And the baseline is GraphWalker.

HPAT sampling optimization: As seen in Figure 11, on average, our HPAT optimization is 812.55× faster than the baseline. The maximum speedup comes from the twitter dataset, where *TEA* retains up to 1,788× speedup. Even the smallest speedup is as high as 5.4×. We also find that the root cause of this speedup variation comes from the difference between the average degrees of various graph datasets. Particularly, since the time complexity of the baseline and

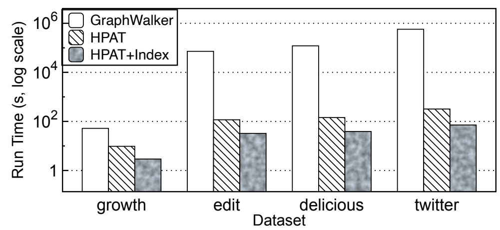
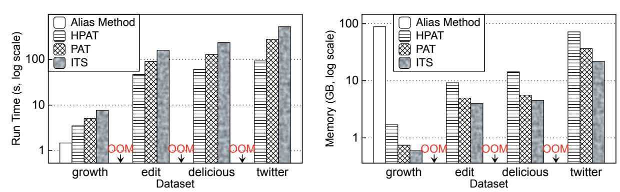


Figure 11. Piecewise breakdown as HPAT and auxiliary index.

TEA are $O(D)$ and $O(log(log(D)) + log(D))$, our sampling is almost insensitive to the degree D. Therefore, for graphs with a higher average degree, like the twitter graph, our speedup climbs.

Auxiliary index optimization further adds 2.75 ~ 3.45× speedup to *TEA*. Recall that the auxiliary index optimization is to help locate the alias table trunks of our interest. This becomes important after our HPAT significantly reduces the sampling time. Similarly, as the first optimization, we observe the biggest gain for twitter from 320.1 to 92.93 seconds while the smallest speedup for growth was from 9.66 to 3.52 seconds. The speedup comes from the time complexity reduction, from $O(log(log(D))+log(D))$ to $O(log(log(D)))$. Note that the impacts of the degree in this optimization are similar to HPAT across datasets.

5.4 Comparison of Various Sampling Methods



(a) Runtime Comparison. **(b)** Memory Usage.

Figure 12. The comparison across various sampling methods, i.e., HPAT, PAT, ITS, and alias method.

This section compares the performance impacts of HPAT and PAT with the traditional Monte Carlo sampling methods, i.e., *inverse transform sampling* [30] and *alias method* [27]. Particularly, for ITS, we can directly use it in *TEA* because the sampling space is organized in time decreasing order that favors ITS sampling space construction. For the alias method, we build multiple versions of alias tables for each possible candidate edge set.

Figures 12a and 12b compare the runtime and memory usage for temporal node2vec respectively. We can see that the alias method has the smallest runtime on the growth dataset but it fails to accommodate other datasets due to overwhelming space consumption (see Section 3.1). Even for the growth dataset, the alias method is a mere 1.38× faster than HPAT but with an astonishing 51.7× larger memory usage than HPAT. For other datasets, HPAT is the fastest sampling method and PAT comes second. Particularly, HPAT can achieve 1.43 ∼ 2.97× speedup than PAT and PAT can achieve 1.22 ∼ 1.89× than ITS. For memory usage, PAT and ITS consume similar memory space, i.e., PAT only consumes, on average, 1.26× larger space than ITS. Further, HPAT consumes, on average, 1.95× larger space than PAT.

5.5 *TEA* Data Structure Construction Cost

This section studies the time consumption of the *TEA* data structure construction (i.e., preprocessing). The preprocessing includes searching candidate edge sets, PAT/HPAT construction, and auxiliary index generation. As discussed in Section 4.2, *TEA* provides a lock-free parallel execution for all these processes. In the following, we show the evaluation of these processes one by one.

Figure 13a shows the evaluation of searching candidate edge sets from a single thread up to 16 threads. With a single thread, it takes 50s (seconds) on the *twitter* dataset and 0.96s on the *growth* dataset. When we scale to 16 threads, it only takes 3.7s on the *twitter* dataset and 0.07s on the smallest dataset. Although the time complexity of this process is $O(|E|log(D))$ with D as the maximum degree number, our multithreaded support helps significantly speed up this ordering process ($O(|E|log(D))$ is reduced to $O(\frac{|E|log(D)}{Threads})$.

Figure 13b presents the evaluation of HPAT construction. Under a single-threaded context, *TEA* takes 233s on the *twitter* dataset while the time consumption shrinks to 18.4s under 16 threads. On the smallest dataset *growth*, *TEA* takes 4.8s under the single-threaded setting and 0.4s under 16 threads. This process has the same time complexity $O(\frac{|E|log(D)}{Threads})$ as the candidate edge sets searching. But this process has bigger constants because it needs to build alias tables with additional constants while searching candidate edge sets only needs an amount of binary search with very small constants. This process takes about 80% of the preprocessing time.

Figure 13c shows the evaluation of auxiliary index generation. This process takes the smallest percentage of the total preprocessing time, only 5%. This is because of the small time complexity, i.e., $O(\sum_{D'=1}^{D} log(D'))$ with D as the maximum degree number. This process takes from 0.025s to 1.1s under 16 threads. Even under the single thread setting, it only needs 12s on the *twitter* dataset. As shown in Table 3, the maximum degree number D of all datasets is up to millions. This is the main reason for the fast auxiliary index generation process.

Figure 13d shows the speedup of our incremental HPAT updating over the naïve baseline that rebuilds the HPAT from scratch. This speedup is affected by two factors, i.e., the batch size of new incoming edges and the vertex degree size. While the first factor is straightforward, the second one could impact the speedup because it decides the workload difference between our method and the naïve one. In this evaluation, we study the batch sizes of 100 and 10000 for vertex degrees of 1, 100, 10k, and 1 million. Generally, when the vertex degree is much smaller than the batch size, the speedup is close to 1. When the batch size is equal to the vertex degree, the speedup is 1.82× and 1.65× under batch size is 100 and 10000 respectively. When the degree size is much larger than the batch size, the speedup is enormous. Particularly, when the degree size is 10^6, the batch size 100 enjoys 8,975× speedup, and the batch size 10000 offers 79.3× speedup.

Figure 13e reports the preprocessing time with respect to the increasing number of threads on the *twitter* dataset. Since the preprocessing step is embarrassingly parallel, we observe close to linear scalability, i.e., 12.8× from 1 to 16 threads. When we perform a random walk of length 80 from each vertex in this dataset, the preprocessing time takes 24% of the total time. This ratio is subject to change when the number and length of the random walks vary.

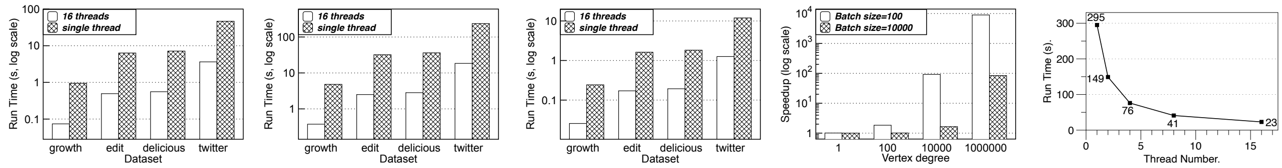
5.6 Studying *TEA* in Out-of-Core Setting

Figure 14a shows the overall runtime of *TEA* and GraphWalker under an out-of-core execution environment, where temporal node2vec is the application. On average, *TEA* is 713× faster than GraphWalker with the maximum and minimum speedups as 1, 172× (*twitter*) and 115× (growth), respectively. Since the out-of-memory setting is closely related to disk I/O, Figure 14b further investigates the I/O performance of *TEA* and GraphWalker. As expected, longer runtime in Figure 14a also experiences longer I/O time in Figure 14b. Particularly, the average, maximum and minimum speedups are 480.4×, 1107.8× (*twitter*), and 130.3× (growth), respectively.

Disk I/O takes the majority of runtime in the out-of-core execution environment. And since we use prefix-sum of edge trunks to pick the trunk of interest for loading, our I/O complexity is hence $O(trunkSize)$ (Section 3.2). For GraphWalker, because it has to load D neighbors in memory for sampling, both its sampling and I/O complexities are $O(D)$. Considering that disk I/O has a remarkably slower speed than CPUs, the computation of sampling takes much less time than neighbor loading from disk. This explains the trend matching between Figure 14b and Figure 14a.

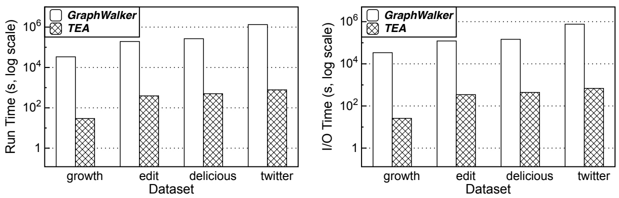
6 Related Work

The majority of the recent random walk applications center around static graph random walk engines. Particularly, DrunkardMob [24] and GraphWalker [40] are high-speed out-of-core random walk engines that target static walks running on a single machine. DrunkardMob loads the selected dataset into the memory in every round to update each random walk's status. GraphWalker optimizes DrunkardMob by using a better data loading strategy and a random walk



(a) Edges Searching. **(b)** HPAT Generation. **(c)** Index Generation. **(d)** HPAT Updating. **(e)** Preprocessing Time.

Figure 13. Preprocessing Breakdown.



(a) Out-of-Core Performance. **(b)** Disk I/O Comparison.

Figure 14. Out-of-Core Execution Analysis.

updating strategy. KnightKing [45] is a distributed random walk engine for *static* graphs. It brings novel optimizations to rejection sampling algorithms. However, these optimizations can neither efficiently address temporal random walks problems nor expedite out-of-core executions. FlashMob [44] and ThunderRW [38] both improve the irregular memory access of random walk while they still can not have an efficient sampling method for temporal graphs with dynamic sampling. C-SAW [32] is a single machine GPU-based random walk engine that aims to optimize ITS sampling algorithms for static graphs. However, none of these sampling optimizations are designed for temporal graphs and thus suffer from high time complexity, and overwhelming space consumption when accommodating temporal graphs.

Temporal graph random walk is a very important type of graph embedding methods [10, 12, 13, 21, 26, 28, 47]. For example, several popular static graph random walk models have been expanded to support temporal graphs, such as node2vec [8, 37, 50, 52]. It is important to note that existing static random walk models are not designed to tackle temporal graph random walk problems. Many current technologies for static random walk models cannot be directly applied to solve temporal graph random walk problems due to being oblivious to the additional time instance dimension. As for random walk models in temporal space, CTDNE [25, 31] proposes the exponential weight random walk which is widely used in temporal graph random walks [17, 41]. CAW [41] proposes *Causal Anonymous Walks* for inductive representation learning in temporal graphs which has a similar edge transition probability to CTDNE but with a different random walk generation strategy. EHNA [17] proposes the temporal node2vec and embeds the temporal random walk algorithm in a stacked LSTM architecture to provide high accuracy but has high programming challenging and even slower execution efficiency than random walk based model CTDNE.

7 Conclusion

This paper presents *TEA*, the first general-purpose random walk engine for temporal graphs. *TEA* proposes a novel method to prevent the edge weight from being affected by dynamic temporal information. Further, we introduce a persistent alias table (PAT), hierarchical persistent alias table (HPAT), and associated auxiliary index mechanism to accelerate the sampling process. Finally, we provide a temporal-centric programming interface for the end users to express various temporal random walk algorithms with ease. Supported by *TEA*, diverse temporal random walk algorithms can benefit from our optimizations. Centered around our novel *persistent alias sampling method*, *TEA* can achieve up to *3 orders of magnitude* performance improvement over the state-of-the-art random walk engines.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd Phillip Stanley-Marbell for their valuable comments and suggestions. The authors from Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This work is supported by National Key Research & Development Program of China (2020YFC1522702), Natural Science Foundation of China (61877035, 62141216), National Science Foundation CRII Award 2000722, NSF 2212370, CAREER Award 2046102, SOAR fellowship, University of Sydney Faculty Startup funding, Australia Research Council (ARC) Discovery Project DP210101984, Ant Group through Ant Research Intern Program, and Tsinghua University Initiative Scientific Research Program. Chengying Huan and Santosh Pandey contributed equally to this research. Hang Liu and Yongwei Wu are joint corresponding authors.

References

[1] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.).

USENIX Association, 125–137. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ai

[2] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: taking the pulse of a fast-changing and connected world. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, Pascal Felber, Frank Bellosa, and Herbert Bos (Eds.). ACM, 85–98. https://doi.org/10.1145/2168836.2168846

[3] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. 257–266. https://doi.org/10.1145/3292500.3330925

[4] 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, WIMS 2017, Amantea, Italy, June 19-22, 2017*. 21:1–21:12. https://doi.org/10.1145/3102254.3102279

[5] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. 2019. A Survey on Network Embedding. *IEEE Trans. Knowl. Data Eng.* 31, 5 (2019), 833–852. https://doi.org/10.1109/TKDE.2018.2849727

[6] 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, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 135–144. https://doi.org/10.1145/3097983.3098036

[7] 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, CIKM 2017, Singapore, November 06 - 10, 2017*. 1797–1806. https://doi.org/10.1145/3132847.3132953

[8] 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, San Francisco, CA, USA, August 13-17, 2016*. 855–864. https://doi.org/10.1145/2939672.2939754

[9] 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*.

[10] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 1024–1034. http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs

[11] Taher H. Haveliwala. 2002. Topic-sensitive PageRank. In *Proceedings of the Eleventh International World Wide Web Conference, WWW 2002, May 7-11, 2002, Honolulu, Hawaii, USA*, David Lassner, David De Roure, and Arun Iyengar (Eds.). ACM, 517–526. https://doi.org/10.1145/511446.511513

[12] Mikael Henaff, Joan Bruna, and Yann LeCun. 2015. Deep Convolutional Networks on Graph-Structured Data. *CoRR* abs/1506.05163 (2015). arXiv:1506.05163 http://arxiv.org/abs/1506.05163

[13] Ryohei Hisano. 2016. Semi-supervised Graph Embedding Approach to Dynamic Link Prediction. *CoRR* abs/1610.04351 (2016). arXiv:1610.04351 http://arxiv.org/abs/1610.04351

[14] Petter Holme and Jari Saramäki. 2012. Temporal networks. *Physics reports* 519, 3 (2012), 97–125.

[15] Chengying Huan, Hang Liu, Mengxing Liu, Yongchao Liu, Changhua He, Kang Chen, Jinlei Jiang, Yongwei Wu, and Shuaiwen Leon Song. 2022. TeGraph: A Novel General-Purpose Temporal Graph Computing Engine. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 578–592. https://doi.org/10.1109/ICDE53745.2022.00048

[16] Chengying Huan, Shuaiwen Leon Song, Yongchao Liu, Heng Zhang, Hang Liu, Charles He, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2022. T-GCN: A Sampling Based Streaming Graph Neural Network System With Hybrid Architecture. In *The 31st International Conference on Parallel Architectures and Compilation Techniques, PACT 2022.*

[17] Shixun Huang, Zhifeng Bao, Guoliang Li, Yanghao Zhou, and J. Shane Culpepper. 2020. Temporal Network Representation Learning via Historical Neighborhoods Aggregation. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1117–1128. https://doi.org/10.1109/ICDE48307.2020.00101

[18] Silu Huang, James Cheng, and Huanhuan Wu. 2014. Temporal Graph Traversals: Definitions, Algorithms, and Applications. *CoRR* abs/1401.1919 (2014). arXiv:1401.1919 http://arxiv.org/abs/1401.1919

[19] 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, July 23-26, 2002, Edmonton, Alberta, Canada*. ACM, 538–543. https://doi.org/10.1145/775047.775126

[20] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: generalized incremental graph processing via graph triangle inequality. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 17–32. https://doi.org/10.1145/3447786.3456226

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=SJU4ayYgl

[22] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. 1269–1278. https://doi.org/10.1145/3292500.3330895

[23] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, Leslie Carr, Alberto H. F. Laender, Bernadette Farias Lóscio, Irwin King, Marcus Fontoura, Denny Vrandecic, Lora Aroyo, José Palazzo M. de Oliveira, Fernanda Lima, and Erik Wilde (Eds.). International World Wide Web Conferences Steering Committee / ACM, 1343–1350. https://doi.org/10.1145/2487788.2488173

[24] Aapo Kyrola. 2013. DrunkardMob: billions of random walks on just a PC. In *Seventh ACM Conference on Recommender Systems, RecSys '13, Hong Kong, China, October 12-16, 2013*. 257–264. https://doi.org/10.1145/2507157.2507173

[25] John Boaz Lee, Giang Hoang Nguyen, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2019. Temporal Network Representation Learning. *CoRR* abs/1904.06449 (2019). arXiv:1904.06449 http://arxiv.org/abs/1904.06449

[26] Ron Levie, Federico Monti, Xavier Bresson, and Michael M. Bronstein. 2019. CayleyNets: Graph Convolutional Neural Networks With Complex Rational Spectral Filters. *IEEE Trans. Signal Process.* 67, 1 (2019), 97–109. https://doi.org/10.1109/TSP.2018.2879624

[27] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. 2014. Reducing the sampling complexity of topic models. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. 891–900. https://doi.org/10.1145/2623330.2623756

[28] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. 2018. Adaptive Graph Convolutional Neural Networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18),*

the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018. 3546–3553. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16642

[29] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. 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*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 135–146. https://doi.org/10.1145/1807167.1807184

[30] Frederic P. Miller, Agnes F. Vandome, and John Mcbrewster. 2010. *Inverse Transform Sampling.*

[31] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-Time Dynamic Network Embeddings. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018.* 969–976. https://doi.org/10.1145/3184558.3191526

[32] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S. Li, and Hang Liu. 2020. C-SAW: a framework for graph sampling and random walk on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 56. https://doi.org/10.1109/SC41405.2020.00060

[33] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014.* 701–710. https://doi.org/10.1145/2623330.2623732

[34] Neil Sarnak and Robert Endre Tarjan. 1986. Planar Point Location Using Persistent Search Trees. *Commun. ACM* 29, 7 (1986), 669–679. https://doi.org/10.1145/6138.6151

[35] Yingxia Shao, Shiyue Huang, Xupeng Miao, Bin Cui, and Lei Chen. 2020. Memory-Aware Framework for Efficient Second-Order Random Walk on Large Graphs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1797–1812. https://doi.org/10.1145/3318464.3380562

[36] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 417–430. https://doi.org/10.1145/2882903.2882950

[37] Guolei Sun and Xiangliang Zhang. 2017. Graph Embedding with Rich Information through Bipartite Heterogeneous Network. *CoRR* abs/1710.06879 (2017). arXiv:1710.06879 http://arxiv.org/abs/1710.06879

[38] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: An In-Memory Graph Random Walk Engine. *Proc. VLDB Endow.* 14, 11 (2021), 1992–2005. http://www.vldb.org/pvldb/vol14/p1992-sun.pdf

[39] John von Neumann. 1951. Various Techniques Used in Connection with Random Digits. In *Monte Carlo Method*, A. S. Householder, G. E. Forsythe, and H. H. Germond (Eds.). National Bureau of Standards Applied Mathematics Series, Vol. 12. US Government Printing Office, Washington, DC, Chapter 13, 36–38.

[40] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. 2020. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 559–571. https://www.usenix.org/conference/atc20/presentation/wang-rui

[41] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. In *International Conference on Learning Representations.* https://openreview.net/forum?id=KYPz4YsCPj

[42] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.

[43] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. 2016. Efficient algorithms for temporal path computation. *IEEE Transactions on Knowledge and Data Engineering* 28, 11 (2016), 2927–2942.

[44] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random Walks on Huge Graphs at Cache Efficiency. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 311–326. https://doi.org/10.1145/3477132.3483575

[45] Ke Yang, Mingxing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. 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.* 524–537. https://doi.org/10.1145/3341301.3359634

[46] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S. Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. *Proc. VLDB Endow.* 13, 5 (2020), 670–683. https://doi.org/10.14778/3377369.3377376

[47] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018.* 974–983. https://doi.org/10.1145/3219819.3219890

[48] Wenchao Yu, Wei Cheng, Charu C. Aggarwal, Kai Zhang, Haifeng Chen, and Wei Wang. 2018. NetWalk: A Flexible Deep Embedding Approach for Anomaly Detection in Dynamic Networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018.* 2672–2681. https://doi.org/10.1145/3219819.3220024

[49] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations.* https://openreview.net/forum?id=BJe8pkHFwS

[50] 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. https://doi.org/10.24963/ijcai.2018/634

[51] Yifeng Zhao, Xiangwei Wang, Hongxia Yang, Le Song, and Jie Tang. 2019. Large Scale Evolving Graphs with Burst Detection. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019.* 4412–4418. https://doi.org/10.24963/ijcai.2019/613

[52] Dongyan Zhou, Songjie Niu, and Shimin Chen. 2018. Efficient Graph Computation for Node2Vec. *CoRR* abs/1805.00280 (2018). arXiv:1805.00280 http://arxiv.org/abs/1805.00280

[53] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *PVLDB* 12, 12 (2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

[54] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[55] Yuan Zuo, Guannan Liu, Hao Lin, Jia Guo, Xiaoqian Hu, and Junjie Wu. 2018. Embedding Temporal Network via Neighborhood Formation. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 2857–2866. https://doi.org/10.1145/3219819.3220054