TEA+: A Novel Temporal Graph Random Walk Engine with Hybrid Storage Architecture

CHENGYING HUAN, Institute of Software, Chinese Academy of Sciences, Beijing, China, Rutgers University, New Brunswick, USA, and Tsinghua University, Beijing, China
YONGCHAO LIU, Ant Group, Hangzhou, China
HENG ZHANG, Institute of Software, Chinese Academy of Sciences, Beijing, China
SHUAIWEN SONG, Sydney University, Sydney, Australia
SANTOSH PANDEY, Rutgers University, New Brunswick, USA
SHIYANG CHEN, Rutgers University, New Brunswick, USA
XIANGFEI FANG, Institute of Software, Chinese Academy of Sciences, Beijing, China
YUE JIN, Ant Group, Hangzhou, China
BAPTISTE LEPERS, Université de Neuchâtel, Neuchatel, Switzerland
YANJUN WU, Institute of Software, Chinese Academy of Sciences, Beijing, China
HANG LIU, Rutgers University, New Brunswick, USA

Many real-world networks are characterized by being temporal and dynamic, wherein the temporal information signifies the changes in connections, such as the addition or removal of links between nodes. Employing random walks on these temporal networks is a crucial technique for understanding the structural evolution of such graphs over time. However, existing state-of-the-art sampling methods are designed for traditional

This article is an extension of conference paper. The preliminary version was published in EuroSys 2023 [16]. In this article, we tackle two main scalability issues: handling dynamic changes in graphs, such as adding or removing edges and vertices, and optimizing performance with varying memory constraints. To solve the first problem, we have developed an efficient graph update method that smartly manages the addition and removal of edges and vertices, maintaining only necessary sampling indices and discarding unnecessary ones. For the second problem, we propose a novel, degree-aware hybrid storage system that optimally utilizes available memory by placing key sampling indices in memory through a fast parallel greedy algorithm, whose efficiency we have theoretically proven. We have conducted comprehensive tests on dynamic graph updates, the parallel greedy algorithm, and performance with limited memory, demonstrating the effectiveness and scalability of our approaches.

We thank the reviewers, AE, and EIC for their invaluable comments and suggestions. This work was supported by Natural Science Foundation of China (61877035, 62141216, 62002350), the Youth Innovation Promotion Association CAS (2023120), National Science Foundation CRII Award 2000722, NSF 2212370, CAREER Award 2046102, a SOAR fellowship, University of Sydney Faculty Startup funding, Australia Research Council (ARC) Discovery Project DP210101984, and the Ant Group through Ant Research Intern Program.

Authors' addresses: C. Huan, Institute of Software, Chinese Academy of Sciences, Beijing, China, Rutgers University, New Brunswick, NJ, USA, and Tsinghua University, Beijing, China; e-mail: huanchengying@iscas.ac.cn; Y. Liu and Y. Jin, Ant Group, Hangzhou, China; e-mails: {yongchao.ly, jinyue.jy}@antgroup.com; H. Zhang (Corresponding author), X. Fang, and Y. Wu, Institute of Software, Chinese Academy of Sciences, Beijing, China; e-mails: zhangheng17@iscas.ac.cn, fangxiangfei58@gmail.com, yanjun@iscas.ac.cn; S. Song, Sydney University, Sydney, Australia; e-mail: shuaiwen.song@sydney.edu.au; S. Pandey, H. Liu (Corresponding author), and S. Chen, Rutgers University, New Brunswick, NJ, USA; e-mails: sp2387@scarletmail.rutgers.edu, hliu77@stevens.edu, shiyang.chen@rutgers.edu; B. Lepers, Université de Neuchâtel, Neuchatel, Switzerland; e-mail: baptiste.lepers@sydney.edu.au.

ACM 1544-3973/2024/05-ART37
https://doi.org/10.1145/3652604

static graphs, and as such, they struggle to efficiently handle the dynamic aspects of temporal networks. This deficiency can be attributed to several challenges, including increased sampling complexity, extensive index space, limited programmability, and a lack of scalability.

In this article, we introduce *TEA+*, a robust, fast, and scalable engine for conducting random walks on temporal graphs. Central to *TEA+* is an innovative hybrid sampling method that amalgamates two Monte Carlo sampling techniques. This fusion significantly diminishes space complexity while maintaining a fast sampling speed. Additionally, *TEA+* integrates a range of optimizations that significantly enhance sampling efficiency. This is further supported by an effective graph updating strategy, skilled in managing dynamic graph modifications and adeptly handling the insertion and deletion of both edges and vertices. For ease of implementation, we propose a temporal-centric programming model, designed to simplify the development of various random walk algorithms on temporal graphs. To ensure optimal performance across storage constraints, *TEA+* features a degree-aware hybrid storage architecture, capable of adeptly scaling in different memory environments. Experimental results showcase the prowess of *TEA+*, as it attains up to three orders of magnitude speedups compared to current random walk engines on extensive temporal graphs.

CCS Concepts: • **Theory of computation → Graph algorithms analysis**; • **Hardware → External storage;**

Additional Key Words and Phrases: Random walk; Graph algorithm; Temporal graph; External Storage

ACM Reference Format:
Chengying Huan, Yongchao Liu, Heng Zhang, Shuaiwen Song, Santosh Pandey, Shiyang Chen, Xiangfei Fang, Yue Jin, Baptiste Lepers, Yanjun Wu, and Hang Liu. 2024. *TEA+*: A Novel Temporal Graph Random Walk Engine with Hybrid Storage Architecture. *ACM Trans. Arch. Code Optim.* 21, 2, Article 37 (May 2024), 26 pages.
https://doi.org/10.1145/3652604

1 INTRODUCTION

Many graphs in real-world scenarios are characterized by their *temporal* and *dynamic* nature, wherein temporal information denotes the timing of changes in connections, such as edge additions and deletions. Such temporal information is often indispensable for accurately interpreting temporal graphs. Figure 1 illustrates the significance of temporal information using a commuting network as an example. In the commuting graph, the formation of a path must adhere to the rule of temporal connectivity. Specifically, when traversing through each vertex, the timestamp of the outgoing edge must be later than that of the incoming edge. For instance, considering the paths reaching vertex 7 from vertex 9, only three paths, 9⟶7⟶4, 9⟶7⟶5, and 9⟶7⟶6, satisfy this criterion. This contrasts sharply with scenarios where temporal information is overlooked. Temporal graphs are subject to dynamic changes over time, experiencing modifications that include the insertion or deletion of both edges and vertices.

For many real-world applications, temporal information is a key metric for extracting valuable insights and making informed decisions. In the following, we enumerate a few more examples, in addition to the aforementioned commute network. In an e-commerce network [20, 47], 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 [20]. A student who has not attended class in the last few days will have a higher probability of dropping out. Educators could intervene proactively to avoid such abrupt career changes. The temporal information in a temporal graph is often indispensable.

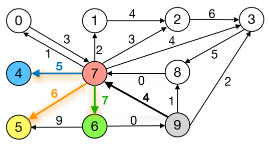


Fig. 1. Commuting network as a temporal graph.

Random walk is a popular and fundamental tool for many graph applications like graph processing, link prediction, graph mining, graph embedding, and node classification [7–9, 11, 13, 14, 30, 34, 42–49]. 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 [13, 30, 42]. This process will continue until it meets certain termination criteria, such as the desired random walk length. Several recent efforts [37, 41, 42] have developed systems to support random walks and their applications on *static graphs* which disregard the temporal information. However, various graph learning projects [17, 28, 39, 51] 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 ensure that the time order of a path consistently increases. Specifically, a temporal graph walker commences from a certain edge, and at each step, it selects an edge with a timestamp greater than the current edge from the out-edges of its current vertex based on a transition probability. In static graphs, this edge sampling step is considered the most challenging aspect of a random walk algorithm [29, 32, 37, 42]. However, the inclusion of temporal information further complicates this process.

First, *rejection sampling* [36], which is often favored for dynamic random walks as suggested by Yang et al. [42], faces significant challenges when applied to temporal random walks due to its high rejection rate. Specifically, in temporal random walks, edge weights frequently incorporate temporal information. For example, an exponential temporal random walk employs the exponential function of the temporal information as the basis for sampling bias (Section 2.3). This leads to a highly skewed probability distribution function (Section 2.2), which in turn drastically narrows the "accept"region. As a consequence, the rejection sampling method undergoes a large number of average trials because of its high rejection rate. This necessitates the consideration of alternative methods, such as **Inverse Transform Sampling (ITS)** [27] or the alias method, for the sampling process.

Considering that the 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. This is mainly because temporal random walks necessitate paths to adhere to an increasing time order, which implies that different walkers may need distinct candidate edge sets even if they sample the same vertex. Take Figure 1 as an example: if a walker enters vertex 7 from vertex 9, the candidate set comprises {4, 5, 6}, whereas entering vertex 7 from vertex 8 results in a candidate set of {0, 1, 2, 3, 4, 5, 6}. In such a scenario, relying solely on the alias method would entail the construction of multiple alias tables, each tailored to different contexts.

Figure 2 presents a comparison of the average sampling cost per step, measured by the number of edges evaluated at each step, between *TEA+* and two recent works across four temporal datasets. KnightKing [42], which is dependent on rejection sampling, demands the evaluation of an average of 11,071 edges per step, which can be attributed to its high rejection rate. Conversely, GraphWalker [37], employing a full-scan sampling technique, generates every edge within the current candidate edge set to create the alias table for each sampling operation. This results in an evaluation of a staggering 19,046 edges per step. Furthermore, when sampling the *twitter* dataset [21], GraphWalker necessitates 1 petabyte of preprocessed data. In stark contrast, *TEA+*, utilizing a hybrid sampling approach (discussed in Section 3.2), only evaluates an average of 5.5 edges per step.

Moreover, there is a notable lack of a comprehensive framework that combines essential algorithmic and system-level optimizations for efficient random walks on temporal graphs. This gap leads to poor user productivity and substandard performance in relevant algorithms and applications. Additionally, as pointed out by KnightKing [42], using walker-centric algorithms in temporal graph frameworks [26, 50] is not intuitive for users and makes tracking walker states difficult. Adding time-related data to random walks makes programming even more complex. Users find it especially challenging to manage the ever-changing sampling space and to develop an effective Monte Carlo sampling strategy for temporal random walk algorithms.

Finally, current state-of-the-art research [16, 42] on temporal graphs faces scalability challenges, especially in scaling to accommodate dynamic graph modifications such as edge insertion/deletion and vertex insertion/deletion. Not only this, but previous work cannot achieve optimal performance across various storage environments. First and foremost, current works [16] typically support only edge insertion, while lacking the capacity to accommodate edge deletions and vertex insertion/deletion. Moreover, these studies struggle to scale efficiently and realize the best performance under varying storage constraints. For instance, in scenarios with a large external storage media, along with varied memory sizes, the performance of existing solutions does not remain optimal.

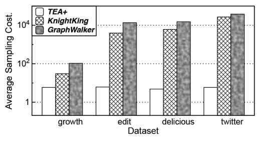


Fig. 2. Average sampling cost defined as #edges/step.

This article presents *TEA+*, a system designed to perform efficiently by using less space, providing fast sampling, offering flexible programming interfaces, and scaling well across different temporal random walk applications. At the heart of *TEA+* is a smart hybrid sampling technique that blends the ITS and alias methods, which significantly reduces space requirements while keeping sampling fast. This method does not depend on the walker's time information for computing edge transition probabilities, and it combines the best of ITS and alias methods by avoiding ITS's slow search operations and the alias method's large space needs. The sampling space is stored in a new data structure called the **Persistent Alias Table (PAT)**. Plus, *TEA+* includes the **Hierarchical Persistent Alias Table (HPAT)** method, supported by an extra index, which improves sampling efficiency. *TEA+* is proficient in handling dynamic graphs, offering robust support for dynamic graph modifications such as the insertion and deletion of both edges and vertices. On the programming side, *TEA+*, built in C++, provides easy-to-use, high-level APIs and lets users design custom functions, enhancing productivity. Additionally, it has a degree-aware hybrid storage architecture that optimizes performance under different memory limitations. Performance tests show that *TEA+* achieves incredible speed, up to 6,158× faster compared to GraphWalker and 954× faster than KnightKing.

2 BACKGROUND

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 [42], 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*—that is, a sequence of all edges that come in the order of time when it is created or collected [20, 40, 49]. In this work, we assume that the temporal graph is updated incrementally. For example, 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.

Dynamic Graph Modification. Temporal graphs experience dynamic changes over time, encompassing updates to both edges and vertices. Edge updates may involve inserting or deleting edges. In a similar vein, vertex updates typically include the insertion or deletion of vertices. It is crucial to note that the modified graph should conform to the edge stream data model. According to this model, newly inserted edges are distinguished by having higher time instances than existing edges, whereas deleted edges have smaller time instances than the live edges.

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 u in a graph $\mathbb{G}$, 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 that the process of selecting a neighbor of the current vertex follows a given probability which is called *edge transition probability* [13, 30, 42]. The edge transition probability for edge (u, v_i) is defined as $P((u, v_i)) = \delta((u, v_i))/\sum_{(u,v_j)\in N(u)} \delta((u, v_j))$, where $\delta((u, v_i))$ is the weight of edge (u, v_i).

Sampling edges is the most time-consuming step in random walk [41, 42]. Here we briefly introduce three sampling methods that are widely used in random walks, including *ITS* [27], the *alias method* [23], and *rejection sampling* [36].

Inverse Transform Sampling. 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 $C[k-1] < r \le C[k]$. Thus, the sampled edge will be the k-th edge in $N(u)$. This process can be executed using a binary search, which has a time complexity of $O(\log(D))$, where D is the degree of u. Using Figure 3(b) 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. The alias method is a technique where the weight of each edge is split into several parts, which are then combined to form n trunks. Essentially, two criteria must be met: (1) each trunk can contain parts from at most two edges, and (2) the total weight in each trunk should be equal to the average weight across all edges. During sampling, a trunk is randomly selected followed by a part within that trunk. Essentially, the probability of selecting an edge is in proportion to the sum of the weights of its parts. Alias tables are the data structures used to store these trunks and their contents. Creating an alias table has a time complexity of $O(n)$, whereas sampling from it has a time complexity of $O(1)$. For example, in Figure 3(c), the alias method creates three trunks with an average weight of 6. During the sampling process, trunk 2 is selected which contains parts from a single edge, leading to the selection of edge $(7, 6, 7)$.

Rejection Sampling. Rejection sampling is a technique [36] that has been recently employed for sampling dynamic edge weights in low-dimensional graphs [42]. The main benefit of rejection sampling is its independence from the need to re-create the sampling space whenever edge weights change, as it considers each edge individually. Once an edge is selected, it is only required to determine if this selection is accepted or rejected. Figure 3(d) shows an example of rejection sampling at vertex 7. Initially, a random number is generated to select a potential edge—for example, vertex 4. Next, another random number is generated to decide whether to accept or reject the sampled edge. If the edge is rejected, the process is repeated until a valid selection is made, such as vertex 6 in Figure 3(d).

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. Next, 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 $P((u, v_i, t_i)) = \delta((u, v_i, t_i))/\sum_{(u,v_j,t_j)\in\Gamma_t(u)} \delta((u, v_j, t_j))$, 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 [28] has implemented this linear weight algorithm to DeepWalk [30].

(II) Exponential Temporal Weight Random Walk. This is another variant of temporal random walk. Using CTDNE [28] 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 (1). The edge transition probability for each edge $(u, v_i, t_i) \subseteq \Gamma_t(u)$ is

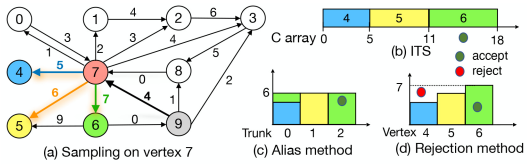


Fig. 3. When a walker arrives at 7 from 9, how different Monte Carlo sampling methods (i.e., ITS, the alias method, and rejection sampling) work on the candidate edge set $(7, 4, 5)$, $(7, 5, 6)$, $(7, 6, 7)$.

$$P((u, v_i, t_i)) = \frac{\delta((u, v_i, t_i))}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} \delta((u, v_j, t_j))} = \frac{exp(t_i - t)}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} exp(t_j - t)} = \frac{exp(t_i)}{\sum_{(u,v_j,t_j)\in\Gamma_t(u)} exp(t_j)}. \tag{1}$$

The exponential temporal weight random walk is widely used in temporal graphs to capture time instances such as CAW [39] 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 [15].

***(III) Temporal Node2vec* [17].** This method extends CTDNE according to the definition of node2vec [13]. 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 $P((u, v_i, t_i)) = \beta_{(u,v_i)} \cdot \delta((u, v_i, t_i))/\sum_{(u,v_j,t_j)\in\Gamma_t(u)} \delta((u, v_j, t_j))$, 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 $d_{(w,v_i)} = 2$.

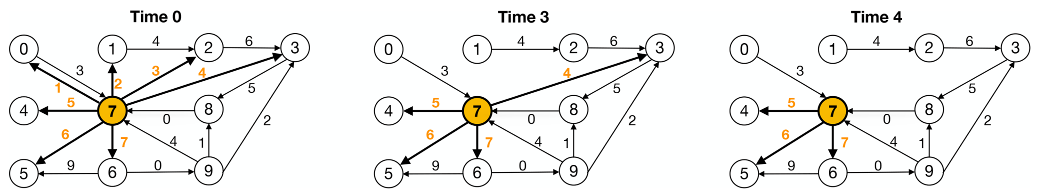


Fig. 4. Evolution of a temporal graph—that is, the candidate edge set and candidate edge weight with time for vertex 7.

The definitions of $\beta_{(u,v_i)}$, $d_{(w,v_i)}$, p, and q are the same as their definitions in node2vec [13] on static graphs. In particular, 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 work, we find that relying solely on any one Monte Carlo sampling algorithm for temporal random walks leads to significant performance issues. Our primary observations are as follows.

First, the rejection sampling method suffers from an exceedingly high number of trials when used on temporal graphs. For instance, in Figure 1, when random walks go from vertex 8 to vertex 7, the candidate set for vertex 7 becomes $\{0, 1, \ldots, 6\}$, with the weight distribution of the exponential temporal weight random walk being $\{exp(1), exp(2), \ldots, exp(7)\}$. This can result in an expected number of trials as high as $\frac{7 \cdot exp(7)}{\sum_{j=1}^{7} exp(j)}$. Figure 2 confirms the high average sampling cost for rejection sampling using KnightKing.

Second, the ITS sampling method maintains a consistent time complexity of $O(\log(D))$, where D denotes the degree of the sampling vertex. Importantly, ITS does not recompute the sampling space for different time ranges, as re-creating the space often takes more time than sampling from the largest range. Therefore, the sampling complexity for ITS remains at $O(\log(D))$.

Third, the alias method needs construct an alias table for each arriving timestamp of every vertex, utilizing pre-computed transition probabilities for efficient sampling. For each vertex v_i, the method stores alias tables for all potential candidate edge sets, each requiring $O(D_{v_i})$ space, where D_{v_i} is the degree of v_i. This results in a space complexity of $O(D_{v_i}^2)$. Consequently, the alias method requires $\sum_{v_i \in V} D_{v_i}^2$ space to store all alias tables. This significant space requirement complicates storing and indexing the alias tables. For example, for vertex 7 in Figure 4, the candidate set for vertex 7 changes based on the arrival time, leading to three distinct alias tables for the same vertex.

Overview. To address the challenges of high sampling cost in rejection sampling, complex time consumption in ITS, and large space requirements in the alias table method for temporal random walks, *TEA+* utilizes a hybrid technique that integrates ITS sampling with the alias method. Additionally, *TEA+* introduces the HPAT design for efficient in-memory sampling, coupled with auxiliary indexing for rapidly locating the needed alias table. Furthermore, *TEA+* is designed with capabilities to support dynamic modifications in graphs, thereby enhancing its utility in scenarios involving dynamic graphs, including the insertion and deletion of edges and vertices.

3.2 Persistent Alias Table

In temporal graphs, candidate edge sets dynamically evolve based on temporal data. Each of these sets can be considered as a combination of several smaller subsets of edges, organized in a time-decreasing sequence. Our PAT method dissects the entire time-decreasing edge set into these smaller, manageable subsets, which we term *trunks*. These trunks are inherently static and a group of trunks are further organized to constitute a dynamic candidate edge set. To navigate through a candidate edge set comprising multiple trunks, we deploy the ITS method, allowing us to select the specific trunk required. The invariant nature of these trunks enables us to efficiently implement the alias method for sampling within any chosen trunk, facilitating effective and efficient sampling even in the dynamic landscape of temporal graphs.

PAT introduces new data structures and algorithms for effective sampling. The key data structure involves the creation of alias tables for each trunk and a prefix-sum array based on these trunks. The construction process is as follows. First, the entire neighbor list of a vertex is divided into equal-sized groups termed *trunks*, each containing a fixed number of edges, ordered by descending time. For instance, as depicted in Figure 5, the neighbor list of vertex 7 is segmented into four trunks: {6, 5}, {4, 3}, {2, 1}, and {0}. Subsequently, an alias table is generated for each trunk, and a prefix-sum array is built across these trunks. Assuming the temporal weight of each edge follows a linear temporal weight rule (illustrated in Figure 5), the alias tables for the trunks could have average values of 6.5, 4.5, 2.5, and 1, respectively. The corresponding prefix-sum array for these trunks would be {0, 13, 22, 27, 28} (refer to the bottom right of Figure 5). This structure is called *PAT*, drawing inspiration from the concept of persistent segment trees.

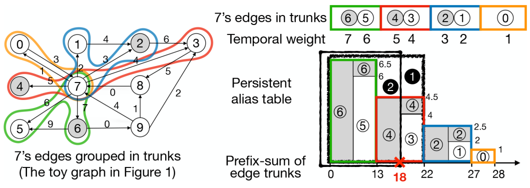


Fig. 5. PAT for vertex 7 of Figure 1 with the edges arranged by decreasing time. Under this new construction, ITS is first used to select a particular trunk, then the alias method is applied to perform sampling inside the selected trunk.

In the PAT data structure, our algorithm addresses two distinct sampling scenarios depending on the completeness of the edges in the trunk selected via ITS where completeness means a candidate edge set is exactly comprised of all edges in the trunks selected. The first scenario involves a situation where all edges in the selected trunk are complete. This case is relatively straightforward, as it allows for direct application of the alias method to sample the desired neighbor within the trunk. An illustration of this scenario is presented in Figure 5 under label ❶, where ❶ denotes the candidate edge set. For instance, consider an incoming edge $(0, 7, 3)$; the candidate set ❶ would then be $\{6, 5, 4, 3\}$. The chosen trunk in this example is complete, with a prefix-sum range extending from 0 to 22. In such cases, the alias method can be directly employed for neighbor sampling.

The second case in the PAT data structure involves sampling within an incomplete trunk. In such cases, the alias method is not applicable, as it requires a complete trunk to function effectively. Instead, ITS is employed for sampling within the trunk, eliminating the need for reconstructing the trunk's alias table. For illustration, consider the candidate edge set ❷ in Figure 5. Suppose the incoming edge is $(9, 7, 4)$; the resulting candidate edge set would be $\{6, 5, 4\}$. This set spans the entire trunk $\{6, 5\}$ and partially includes the trunk $\{4, 3\}$, excluding edge 3. In scenarios like this, PAT constructs the prefix-sum array for the incomplete trunk, such as the prefix-sum array for the edge set $\{4\}$, and subsequently performs ITS on this array to facilitate sampling.

Intuitively, our PAT method addresses the limitations inherent in both the alias table and ITS methods. First, compared to the conventional alias method design, for each vertex u, we achieve a reduction in space consumption from $O(D^2)$ to $O(D)$, where D denotes the degree of vertex u. The alias method incurs a significant space cost of $O(D^2)$, primarily because it requires constructing D different versions of alias tables for each vertex. Each of these versions caters to one of the D possible candidate edge sets, with each alias table demanding $O(D)$ space. Hence, the cumulative space complexity for maintaining all of these alias tables per vertex escalates to $O(D^2)$. In contrast, our PAT method maintains efficiency with $O(D)$ space: the alias table trunks themselves require $O(D)$ space, whereas the prefix-sum of trunks needs only $O(\frac{D}{\text{trunkSize}})$ space. Second, when contrasting with the ITS method design, our PAT approach optimizes the searching time complexity from $O(\log(D))$ to $O(\log(\frac{D}{\text{trunkSize}}))$. This optimization stems from the reduced range in the prefix-sum array due to the division into trunks, thereby expediting the search process.

3.3 Hierarchical Persistent Alias Table

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

Equations (2), (3), and (4) formally define how we construct our HPAT for each vertex u with edge set as $\{e_1, \ldots, e_n\}$ in a hierarchical manner:

$$\tau_u = \{\tau_u^0, \ldots, \tau_u^k, \ldots, \tau_u^K\},\ 0 \le k \le K = \lfloor \log_2(|N(u)|) \rfloor, \tag{2}$$

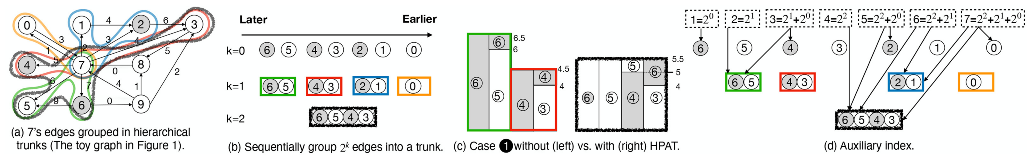
$$\tau_u^k = \{\tau_u^{k,0}, \ldots, \tau_u^{k,i}, \ldots, \tau_u^{k,I}\},\ 0 \le i \le I = \left\lfloor \frac{|N(u)|}{2^k} \right\rfloor - 1, \tag{3}$$

$$\tau_u^{k,i} = \{e_{i\cdot 2^k+1}, \ldots, e_{(i+1)\cdot 2^k}\}. \tag{4}$$

In Equation (2), each τ_u^k represents a relatively larger trunk. Subsequently, in Equation (3), we divide each of the larger trunks, τ_u^k from Equation (2), into smaller trunks denoted by $\tau_u^{k,i}$. Moreover, as demonstrated in Equation (4), each trunk is represented by an edge set of $\tau_u^{k,i}$, which signifies the i-th trunk of vertex u comprising 2^k edges. For every trunk, whether larger or smaller, we construct an alias table for subsequent sampling. During the sampling process, since each candidate edge set $\{e_1, \ldots, e_i\}$ must be a prefix of the current vertex's edge set $\{e_1, \ldots, e_n\}$ arranged in descending order of time, the candidate edge set can be partitioned into a series of trunks through binary decomposition. Initially, *TEA+* samples these trunks using *ITS*, which helps in reducing the overall space overhead. Subsequently, the alias table associated with the sampled trunk is employed to locally sample an edge. This is where the *alias method* is utilized to facilitate rapid sampling.

Continuing with vertex 7 from Figure 1 as an example, the candidate set is $\{6, \ldots, 0\}$. Figure 6(b) shows our trunk sets, where τ_u can be represented as $\tau_u^0 = \{\{6\}, \ldots, \{0\}\}$, $\tau_u^1 = \{\{6,5\},\{4,3\},\{2,1\}\}$, and $\tau_u^2 = \{\{6,5,4,3\}\}$. During sampling, the candidate set is divided into three trunks ($7 = 2^2 + 2^1 + 2^0 = 4 + 2 + 1$), namely `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 $g_3 = \{0\} = \tau_u^{0,6}$. The sampled probabilities of each trunk can then be calculated using the *ITS* array C: $P(g_1) = (0, \frac{C[4]}{C[7]}]$, $P(g_2) = (\frac{C[4]}{C[7]}, \frac{C[6]}{C[7]}]$, and $P(g_3) = (\frac{C[6]}{C[7]}, 1]$. Once all available trunks have been sampled through *ITS*, the local alias method is employed to sample an edge within the selected trunk.

This design further reduces the time complexity of PAT to $O(\log(\log(D)))$, as each candidate edge set encompasses up to $\log(D)$ trunks. Subsequently, local processing using the alias method within each sampled trunk takes a constant time, $O(1)$. In terms of space consumption, only the alias tables of the subsets of $\tau_u^{k,i}$ need preprocessing, which results in a space overhead of D for τ_u^k and an overall space overhead of $O(D\log(D))$ when D represents the degree number. This approach is substantially more efficient compared to the direct application of the alias method for random walks on temporal graphs, which requires a space complexity of $O(D^2)$. Although the space overhead is higher than that of *ITS*, which requires only D space, our sampling methods are faster than it, $O(\log(\log(D)))$ vs. $O(\log(D))$.



Later
Earlier
k=0
k=1
k=2
(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.

Fig. 6. HPAT design 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. (d) The auxiliary index for HPAT.

We have implemented two specific *ad hoc* optimizations in our approach. The first optimization involves disregarding certain neighbors if their temporal information precedes that of all incoming edges. The second optimization applies when a vertex has a relatively low out-degree; in such cases, we construct alias tables exclusively for that vertex's out-edges.

3.4 Auxiliary Index

This section introduces the concept of the auxiliary index, designed to reduce the time complexity of finding the trunks of interest for each candidate edge set from $O(\log(D))$ to $O(1)$. In the sampling process, HPAT designs necessitate identifying the trunks that contain the candidate edges, denoted as $\Gamma_t(u)$, which would otherwise result in a time complexity of $O(\log(D))$. We provide a detailed complexity analysis in the following. When a walker arrives at vertex u at time t, *the objective is to find the minimum number of trunks, each of size 2^i, that can collectively construct* $\Gamma_t(u)$. This process requires $\log(|\Gamma_t(u)|)$ operations to decompose $|\Gamma_t(u)|$ into trunks of appropriate sizes, and an additional $\log(D)$ operations to locate these trunks. For example, consider vertex 7 in Figure 1: when a walker arrives at vertex 7 from vertex 0, the candidate edge set is $\Gamma_{t=3}(7) = \{6, 5, 4, 3\}$. For HPAT, the required trunk is $\{6, 5, 4, 3\}$. Conversely, if vertex 7 is reached from vertex 9, $\Gamma_{t=4}(7)$ would be $\{6, 5, 4\}$. In this scenario, the trunks of interest for the hierarchical persistent alias method are $\{6, 5\}$ and $\{4\}$.

Our auxiliary index enables *TEA+* to rapidly identify the trunks of interest. Figure 6(d) illustrates *how to build an auxiliary index* for the HPAT of vertex 7. Assuming that the HPAT requires seven neighbors from vertex 7, as indicated by the top right dotted box in Figure 6(d), these neighbors are distributed across three trunks: $\{6, 5, 4, 3\}$, $\{2, 1\}$, and $\{0\}$. Since HPAT organizes all alias tables into a complete binary search tree, with indices $2^2 + 2^1 + 2^0$=4 + 2 + 1, we can locate the trunks of interest as follows. First, the value $2^2 = 4$ directs us to fetch the only size 4 trunk at the bottom level (i.e., $\{6, 5, 4, 3\}$). Second, the value $2^1 = 2$ indicates that our second trunk of interest is in the second level of the binary search tree, starting from position 4, which is the cumulative size of the previous trunk. Finally, the value $2^0 = 1$ signifies that this trunk is in the third level (where each trunk's size is 1), and its position is the sum of the sizes of previously fetched trunks (i.e., $4+2 = 6$). Thus, we obtain the last trunk. This design significantly reduces the time complexity of finding trunks from $O(\log(D))$ to $O(1)$.

3.5 Dynamic Graph Modification

Dynamic graph modifications typically include the insertion and deletion of both edges and vertices. These updates are carried out in batches, a method akin to those employed in state-of-the-art dynamic graph engines [6, 19, 33]. To support these dynamic graph updates, *TEA+* adapts by appropriately adding to or removing from the sampling index.

Edge Insertion. For each batch of edge insertion, we need to update the PAT and HPAT indices. Fortunately, both of 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 in the following. 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 6(b). Assume that 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.

Edge Deletion. For conventional evolving graphs, the deletion of edges usually incurs a higher time cost for index updating [3] compared to edge insertion. When confronting edge deletion, the straightforward approach of directly removing an edge necessitates the complete reconstruction of the HPATs. This process is markedly resource-intensive. For instance, when deleting the edge from vertex 7 to $\{0, 1, 2, 3\}$ in Figure 1, the trunks of $k = 1$ must be altered to $\{6, 5\}$ and $\{4\}$, and the trunk of $k = 2$ will become empty. As a result, each deletion operation requires the rebuilding of HPATs. However, because edge deletion adheres to the time increase of edges (i.e., the deleted edges have smaller timestamps than the remaining edges), we can employ an incremental approach to retain the existing alias tables and eliminate the unnecessary ones. Specifically, we only need to remove the alias tables that contain the deleted edges. For example, in the case of deleting the edge from 7 to $\{0, 1\}$, the trunks of $k = 0$ containing $\{0\}$ and $\{1\}$ will be eliminated, and the trunk of $k = 0$ containing $\{2, 1\}$ will also be removed. Subsequently, the remaining trunks can still form a complete HPAT, and sampling on the updated HPAT can yield an accurate result because the set of candidate edges and their binary decomposition originate from the edge with the largest timestamp, whereas the deleted edges only possess the smallest timestamps. Consequently, this edge deletion method can obtain the correct result. The time complexity of each edge deletion is only $O(log(D))$, which is sufficiently low since there are at most $log(D)$ trunks containing the edge.

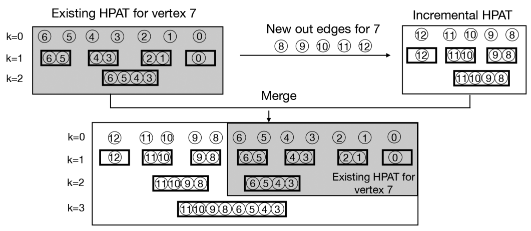


Fig. 7. HPAT updating for vertex 7 of Figure 1 with five new edges from 7 to {8, 9, 10, 11, 12}, respectively.

Vertex Insertion/Deletion. In *TEA+*, vertex addition and removal are essentially extensions of edge operations, as the sampling index is constructed based on edges. Specifically, adding a vertex is akin to inserting a batch of edges in terms of index updating. Similarly, removing a vertex parallels the process of deleting a batch of edges in the index update mechanism.

Dynamic Graph Structure. Dynamic changes in the graph structure over time can be represented as a series of batch operations involving the insertion and deletion of edges and vertices. Following these structural changes, the degree distribution of the graph significantly influences the performance of *TEA+*. Graphs with a high degree distribution tend to exhibit enhanced performance in *TEA+*, primarily due to its advantage in sampling time complexity, which improves from $O(D)$ to $O(\log(\log(D)))$. Consequently, a higher degree number distribution typically leads to a more pronounced speedup in *TEA+*'s performance.

3.6 Sampling Complexity Analysis

This section compares the sampling time complexities of *TEA+* with those of state-of-the-art solutions, GraphWalker [37] and KnightKing [42].

TEA+ always employs HPAT. In the case of linear temporal weight random walk, both GraphWalker and KnightKing utilize the ITS method. For the exponential variant, GraphWalker calculates the probability distribution sequentially before sampling the chosen edge, whereas KnightKing uses rejection sampling. Regarding temporal node2vec, GraphWalker employs a full-scan sampling method for the dynamic weight component (refer to Section 1), whereas KnightKing again employs rejection sampling. Both of them use rejection sampling for the dynamic parameter component.

In the linear temporal weight random walk, GraphWalker and KnightKing both exhibit a time complexity of $O(\log(D))$. In contrast, *TEA+* achieves a lower time complexity of $O(\log(\log(D)))$ with the assistance of HPAT. For the exponential temporal weight random walk, GraphWalker has a time complexity of $O(D)$ due to its sequential pass, whereas KnightKing's time complexity is $O\left(\frac{1}{\varepsilon}\right)$, with ε being the acceptance ratio in rejection sampling. This ratio is calculated as $\varepsilon = \frac{\delta(e_1)+\delta(e_2)+\ldots+\delta(e_D)}{D\cdot\delta(e_D)}$, where $\delta(e_i)$ represents the weight of edge e_i and $\delta(e_D)$ is the maximum weight among these edges. As explained in Section 3.1, the acceptance ratio is always greater than $\frac{1}{D}$, making KnightKing's time complexity marginally less than $O(D)$. Meanwhile, *TEA+* consistently maintains a time complexity of $O(\log(\log(D)))$. Last, for temporal node2vec, which incorporates a dynamic parameter β into the exponential temporal weight random walk, all of these engines employ rejection sampling since β necessitates a small and constant expected number of trials.

4 DEGREE-AWARE HYBRID STORAGE ARCHITECTURE

When facing varying memory sizes, efficiently allocating HPATs and PATs across different vertices is key to optimizing memory usage. This involves keeping a certain number of HPATs and PATs in memory while storing others on disk. HPATs, although faster in sampling, demand more space and disk I/O, as detailed in Section 3. Specifically, HPATs require $O(D\log(D))$ space, more than PAT's $O(D)$, where D is the degree number of each vertex. This larger space need results in increased disk I/O—for instance, HPAT requires $O(D\log(D))$ for each sampling, whereas PAT only needs $trunkSize$, as it samples trunks in memory and loads each trunk from disk as needed. However, HPAT has a lower sampling complexity of $O(\log(\log(D)))$ compared to PAT's $O(\log(\frac{D}{trunkSize}))$. To balance rapid sampling speed with minimal space overhead, we selectively store certain HPATs and PATs in memory, with the remainder on disk, maximizing memory use while maintaining fast sampling speeds. Note that the time analysis in the following section is based on estimated times from the time complexity, not formal time complexity analysis.

4.1 Dynamic Programming For Index Seletion

The process of selecting HPAT/PAT closely resembles the *knapsack problem*, in which the memory serves as the "knapsack."In this scenario, we are required to select specific HPATs and PATs to store in the "knapsack"(memory) with the aim of minimizing execution time. In this context, the overall time T can be expressed as $T = \sum_u(T_uX_u)$, where $T_u = \frac{log(log(D_u))}{C}$ if $\delta_u = 0$, $\frac{log(D_u/trunkSize)}{C}$ if $\delta_u = 1$, $\frac{log(log(D_u))}{C} + \frac{D_u log(D_u)}{B}$ if $\delta_u = 2$, and $\frac{log(D_u/trunkSize)}{C} + \frac{trunkSize}{B}$ if $\delta_u = 3$. X_u denotes the total number of times vertex u is sampled across various random walk applications.

In this formulation, D_u represents the degree number of vertex u, $trunkSize$ signifies the trunk size, B stands for the disk's reading speed, C is the CPU computation speed, and X_u indicates the sampling times of vertex u. Finally, δ_u is used to denote the type of vertex u, where $\delta_u = 0$ means that the HPAT of u is stored in memory, $\delta_u = 1$ means that the PAT of u is stored in memory, $\delta_u = 2$ signifies that the HPAT of u is stored on the disk, and $\delta_u = 3$ means that the PAT of u is stored on the disk.

This equation has a constraint that the size of the HPAT/PAT stored in memory must not exceed the total available memory size. More specifically, the overall memory size $M = \sum_u M_u$ must be smaller than or equal to the available memory size, where $M_u = D_u log(D_u)$ if $\delta_u = 0$, and D_u if $\delta_u = 1$. The objective is to minimize T while adhering to the memory limit constraints. The basic strategy to accomplish this involves the use of dynamic programming as outlined next:

$$dp[u][m] = \begin{cases} min\{dp[u][m], dp[u-1][m - D_u log(D_u)] + \frac{log(log(D_u))}{C} X_u\}, \delta_u = 0, & (5) \\ min\{dp[u][m], dp[u-1][m - D_u] + \frac{log(\frac{D_u}{trunkSize})}{C} X_u\}, \delta_u = 1, & (6) \\ min\{dp[u][m], dp[u-1][m] + (\frac{log(log(D_u))}{C} + \frac{D_u log(D_u)}{B}) X_u\}, \delta_u = 2, & (7) \\ min\{dp[u][m], dp[u-1][m] + (\frac{log(\frac{D_u}{trunkSize})}{C} + \frac{trunkSize}{B}) X_u\}, \delta_u = 3, & (8) \end{cases}$$

where $dp[u][m]$ represents the minimum execution time up to vertex u—that is, from vertex 1 to u, when the memory usage is m. The $dp[u][m]$ is initialized to positive infinity, denoted as $dp[u][m] = +\infty$. The time complexity of this dynamic programming approach is $O(|V|M)$, where V represents the set of vertices. Given that memory size is typically large—for example, $M = 32GB \approx 3.4 \times 10^{10} B$—this dynamic programming method can induce considerable overhead. Additionally, the space complexity is $O(|V|M)$, which can lead to substantial memory overhead.

4.2 Greedy Algorithm Optimization

In practice, much of the search space in dynamic programming is superfluous—that is, it does not contribute to the final optimal solution. Specifically, a significant portion of the function $dp[u][m]$ search space is redundant and can be avoided. To prune this search space, we make three key observations to reduce time complexity based on the fact that the I/O speed B is significantly slower than the computation speed C.

Observation 1. *HPAT should be completely stored in memory. The justification for this is that the increase in sampling performance provided by HPAT outweighs the slowdown incurred by the I/O time necessary to load HPAT. This is substantiated in depth in Theorem 1, suggesting that PATs should be the only structures stored on disks.*

Theorem 1. *For a vertex u that is stored on disk, its PAT can achieve better sampling performance compared to HPAT.*

Proof. For HPAT of vertex u, the disk I/O time is $\frac{Dlog(D)}{B}$ due to the necessity of loading the entire HPAT into memory, and the sampling time is $\frac{log(log(D))}{C}$. Therefore, the sampling time of HPAT is $T_{HPAT} = \frac{Dlog(D)}{B} + \frac{log(log(D))}{C}$. In contrast, for PAT, the sampling time is $T_{PAT} = \frac{trunkSize}{B} + \frac{log(\frac{D}{trunkSize})}{C}$. The time difference $T_{diff} = T_{HPAT} - T_{PAT}$ is

$$T_{diff} = \frac{Dlog(D) - trunkSize}{B} - \frac{log(\frac{D}{trunkSize}) - log(log(D))}{C}. \quad (9)$$

Because B is usually smaller than C, we can derive the following equation:

$$\begin{aligned} T_{diff} &> \frac{Dlog(D) - trunkSize}{B} - \frac{log(\frac{D}{trunkSize}) - log(log(D))}{B} \\ &= \frac{Dlog(D) - log(D) + log(log(D)) - trunkSize + log(trunkSize)}{B} > \frac{(D-1)log(D) - trunkSize}{B}. \end{aligned} \tag{10}$$

Because D is usually larger than $trunkSize$ (i.e., $D > trunkSize$), it follows that $D - 1 \geq trunkSize$. Additionally, since $log(D)$ is typically much larger than 1, we can infer the following:

$$T_{diff} > \frac{D - 1 - trunkSize}{B} \geq 0. \tag{11}$$

This implies that HPAT takes more time than PAT when stored on disk. □

Observation 2. *If the HPAT size of a vertex, which is stored in memory, is larger than the sum of its PAT size and the PAT size of any other vertex, this vertex should substitute its HPAT with a PAT. This suggests that HPATs should be employed mainly for vertices with smaller degrees, since vertices with higher degrees would utilize an excessive amount of memory, resulting in a large number of vertices being stored on disk. This is further elaborated in Theorem 2.*

Theorem 2. *For each vertex u that utilizes HPAT, optimal performance is achieved when there is no other vertex v using PAT such that the size of PAT for u is smaller than the difference between the HPAT size of u and the PAT size of u.*

Proof. We will prove this by contradiction. Assume that u utilizes HPAT, and there exists another vertex v satisfying the aforementioned condition. In this scenario, the total time consisting of the current sampling time plus the I/O time, denoted as T_1, is given by

$$T_1 = \frac{log(log(D_u))}{C} X_u + \frac{log(\frac{D_v}{trunkSize})}{C} X_v + \frac{trunkSize}{B} X_v. \tag{12}$$

Next, if we substitute the HPAT of u with a PAT, then both PATs of u and v can be accommodated in memory, eliminating the need for additional disk I/O. In this situation, the total time including sampling time and I/O time is denoted as T_2:

$$T_2 = \frac{log(\frac{D_u}{trunkSize})}{C} X_u + \frac{log(\frac{D_v}{trunkSize})}{C} X_v. \tag{13}$$

The difference in time is then given by

$$\begin{aligned} T_1 - T_2 &= \frac{log(log(D_u))}{C} X_u - \frac{log(\frac{D_u}{trunkSize})}{C} X_u + \frac{trunkSize}{B} X_v \\ &= \frac{trunkSize}{B} X_v - \frac{log(\frac{D_u}{trunkSize \cdot log(D_u)})}{C} X_u > \frac{trunkSize}{B} X_v - \frac{trunkSize}{C} X_u. \end{aligned} \tag{14}$$

In real-world applications, $log(\frac{D_u}{trunkSize \cdot log(D_u)})$ is typically less than trunkSize. Moreover, the ratio $\frac{C}{B}$ generally exceeds $\frac{X_u}{X_v}$, leading to $\frac{X_v}{B} > \frac{X_u}{C}$. As a result, we can deduce that $T_1 > T_2$. This implies that T_1 does not represent the most efficient performance, and this contradiction refutes the initial assumption. Consequently, the theorem is proven. □

Observation 3. *For two vertices both utilizing PATs, the vertex with the higher degree should be stored in memory. This reasoning is intuitive since vertices with higher degrees are more likely to be revisited [41]. A detailed proof of this can be found in Theorem 3.*

Theorem 3. *For vertices u and v, both employing PATs, if memory can accommodate only one of them, the vertex with the higher degree will be stored in memory.*

Proof. Let us assume that vertex u has a higher degree. The execution time T_1 when storing u in memory and v on disk is

$$T_1 = \frac{log(\frac{D_u}{trunkSize})}{C} X_u + (\frac{log(\frac{D_v}{trunkSize})}{C} + \frac{trunkSize}{B}) X_v. \tag{15}$$

The execution time T_2 when storing u on disk and v in memory is

$$T_2 = \left(\frac{log(\frac{D_u}{trunkSize})}{C} + \frac{trunkSize}{B} \right) X_u + \frac{log(\frac{D_v}{trunkSize})}{C} X_v. \tag{16}$$

Then $T_2 - T_1 = \frac{trunkSize}{B}(X_u - X_v)$. Because u has a higher degree, which results in a larger X_u, we can conclude that $T_2 > T_1$. Thus, storing the PAT of u in memory yields better performance. □

Based on Observation 1, we keep HPATs in memory. In line with Observation 2, we focus on vertices with the smallest degrees while generating HPATs. As per Observation 3, we target vertices with the highest degrees when creating PATs, which are stored in memory. These observations enable us to address the knapsack problem through an efficient greedy algorithm as an alternative to the high-overhead dynamic programming approach. Specifically, the greedy algorithm is composed of the following three steps:

— *Step 1*: Start by enumerating vertices with the smallest degrees, generate HPATs for them, and keep them in memory.
— *Step 2*: For each HPAT that has been enumerated, determine the maximum number of vertices for which PATs can be accommodated in memory.
— *Step 3*: For each scenario considered, calculate the total execution time and compare it to the best-case scenario to make an optimal decision.

We employ Step 1 and Step 2 to explore various HPAT/PAT allocation strategies. As depicted in Figure 8, we initially arrange the vertices in descending order based on their degree. Then, in Step 1, we identify which vertices will be allocated HPATs (i.e., vertices from v_j to v_n), ensuring that these are stored in memory. Subsequently, using Step 2, we determine the maximum number of vertices for which PATs can be generated and accommodated in memory (i.e., vertices from v_1 to v_i). The rest of the vertices (i.e., vertices from v_{i+1} to v_{j-1}) are designated for PATs and are stored on the disk. Finally, Step 3 helps in calculating the total execution time for each allocation strategy and in selecting the most efficient one.

Trunk Size Selection. For PATs kept in memory, the goal is to maximize $trunkSize$ while making sure that the time complexity of ITS on the prefix-sum of trunks (i.e., $O(log(\frac{D}{trunkSize})))$ remains greater than or equal to the time complexity of ITS within each trunk ($O(log(trunkSize))$). This implies that $trunkSize$ should not go beyond $\sqrt{D}$. For PATs stored on disks, it is best to keep $trunkSize$ as small as feasible, provided that there is enough memory to accommodate the prefix-sum array of all trunks, which has a size of $\frac{D}{trunkSize}$. During each iteration in Step 1, we can calculate the optimal $trunkSize$ for the best performance.

Complexity Analysis. The time complexity of the greedy algorithm is clearly $O(|V|log(|V|))$, where $log(|V|)$ arises from Step 2 due to binary search. In comparison to dynamic programming (refer to Section 4.1), the greedy algorithm prunes much of the unnecessary search space, leading to a substantial performance boost. Moreover, unlike dynamic programming which requires $O(|V|M)$ space for storing the outcomes of all allocation schemes, the greedy algorithm only necessitates a $O(|V|)$ space complexity to keep the optimal result.

Parallel Greedy Algorithm. Given that each index allocation scheme's search is independent of the others, we can allocate each thread to calculate the time of each allocation scheme. Each thread will record an optimal answer. After running concurrently, all threads aggregate their answers, resulting in minimal overhead. The time complexity is further reduced to $O(\frac{|V|log(|V|)}{thread\_num})$, with $thread\_num$ as the number of threads.

5 SYSTEM IMPLEMENTATION

TEA+ extends the walker-centric concept on KnightKing to our temporal-centric one which lets users think from the "time" perspective.

Temporal-Centric API, 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: parameters and subgraph selection. Parameters (e.g., *Dynamic_weight* and *Dynamic_parameter*) allow users to customize the bias according to different applications. Subgraph selection provides more expressiveness to users—that is, *Edges_interval. Edges_interval* derives the subgraphs of interest for *TEA+* to perform random walk. All of 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 [28]. *Dynamic_parameter* defines the parameter $\beta_{(u,v_i)}$ of temporal node2vec. *Edges_interval* is provided for users to generate a subgraph (snapshot) for random walk according to different applications.

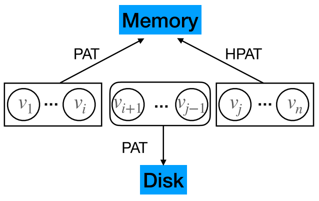


Fig. 8. Greedy algorithm of index selection.

Regarding the interaction between APIs and the *TEA+* framework, initially, *TEA+* makes use of *Edges_interval* to extract the subgraph pertinent to each query. Subsequently, the framework uses a preprocessing function to generate alias tables and an auxiliary index. During the generation of the random walk, the sampling function leverages HPAT to encode both *Dynamic_weight* and the random number R, which is produced by a random number generator. Then *TEA+* uses the rejection sampling to check whether this sampling trial is in the "Accepted" area—that is, 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.

ALGORITHM 1: Temporal node2vec example.

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

Dynamic_parameter (Vertex u, Vertex v)
    if(u==v)
        return $\frac{1}{p}$
    else if(u.ISNEIGHBOR(v))
        return 1
    else
        return $\frac{1}{q}$

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]}
```

Parallel Index Construction. *TEA+* primarily consists of the following three components: (1) searching the candidate edge set for each edge, (2) constructing the PAT/HPAT for each vertex, and (3) generating the auxiliary index for the HPATs. Fortunately, all of these steps can be executed in parallel. For a detailed statement of this process, please refer to our preliminary work [16].

6 EVALUATION

6.1 Experimental Setup

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. Whereas *TEA+* is evaluated in both single and distributed machines, 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 Xeon CPU E5- 2640 v2 @ 2.00 GHz (each has eight cores), 94 GB of DRAM (20-MB L3 cache), and a 1-TB SATA SSD (650 MB/s for sequential read). For distributed machines, we use an eight-node cluster (each node is the same as our single-machine configuration) with 40-Gbps IB interconnection.

Benchmarks. Table 1 provides details of the datasets used in this section. We have selected four widely used datasets from the Koblenz Large Network Collection [21], all of which are temporal graphs in the standard edge stream format. For each graph, $|V|$ represents the number of vertices, $|E|$ signifies the number of edges, *Degree Mean* denotes the average number of edges connected to each node, and *Max Degree* indicates the maximum degree among all nodes in the graph. Notably, these datasets, especially the larger ones, exemplify power-law graphs.

Table 1. Graph Datasets ($k = 10^3$)

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	1,468,365k	74.678	3,691,240
RMAT1	16,777k	536,871k	32.000	2,594
RMAT2	67,108k	1,073,741k	16.000	1,827

Walker 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 [42]. 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, 42].

Baselines. For the all-in-memory mode, we compare *TEA+* with both GraphWalker [37] and KnightKing [42]. We directly use open source codes of GraphWalker and KnightKing. KnightKing uses an eight-node distributed setting which has better performance than only on a single node [42]. Note that both GraphWalker and KnightKing use binary search to search candidate edge sets on sampling, whereas *TEA+* does not. For the external-memory mode, we only compare *TEA+* to GraphWalker because GraphWalker can support external memory well, whereas KnightKing cannot.

6.2 *TEA+* vs. State-of-the-Art Systems

Table 2 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 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.

Exponential Temporal Weight Random Walk. Due to the dynamic edge weights in the exponential temporal weight random walk, GraphWalker has to regenerate the transition probabilities on-the-fly. This necessitates scanning all neighbors, making GraphWalker take around 62.3 hours on the largest dataset, twitter. However, *TEA+* completes the entire sampling process in just 1.2 minutes, achieving a remarkable speedup of up to 3,140× across various settings and at least 13.6× even on the smallest dataset. KnightKing performs better than GraphWalker despite the skewness in probability distribution due to the exponential function, which could lead to numerous trials. This is because the number of trials in KnightKing is always less than the degree of nodes, which affects the sampling overhead in GraphWalker. Additionally, since KnightKing is

Table 2. Runtime Performance (in Seconds) and Speedup of *TEA+* over GraphWalker and KnightKing

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

a distributed system operating on an eight-node cluster, it gains a significant speed advantage over GraphWalker. However, *TEA+* outshines KnightKing as well by achieving up to 531× speedup.

Temporal Node2vec. Different from the exponential random walk, temporal node2vec further adds the dynamic parameter β into the former's random walk generation and the time complexity of temporal node2vec is close to β as shown in Section 3.6. 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 [42]. For each trial, temporal node2vec runs an exponential random walk. As large degree vertices tend to have higher trial numbers and *TEA+* can achieve better performance on the large degree numbers, *TEA+* has better improvement on temporal node2vec than the exponential random walk. In particular, the speedup of *TEA+* is $14.84 \sim 6,158\times$ over GraphWalker and up to 954× 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—that is, $82.5\% \sim 91.2\%$, of the total memory usage. Compared with state-of-the-art systems, GraphWalker takes 36.48 GB on twitter, whereas KnightKing takes a maximum of 6.91 GB per node under eight-node distributed execution. When KnightKing is executed in a single node, it takes 45 GB on twitter. While HPAT takes a slightly larger space, it is astonishingly 6,158× and 954× faster than GraphWalker and KnightKing on this twitter graph, respectively.

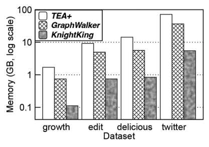


Fig. 9. Memory usage.

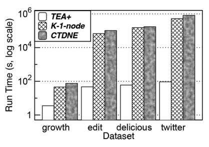


Fig. 10. *TEA+* VS others.

Comparison with Other Engines. Figure 10 compares *TEA+* with other engines, including KnightKing under the single node execution (K-1-node) and CTDNE [28] on temporal node2vec random walk. The general trend is that our *TEA+* outperforms both K-1-node and CTDNE tremendously. In particular, *TEA+* can achieve $5,627\times$ speedup over K-1-node. Compared to CTDNE, *TEA+* can achieve up to $8,816\times$ speedup because CTDNE provides a temporal graph random walk based neural network model rather than an efficient random walk system with system-level optimizations.

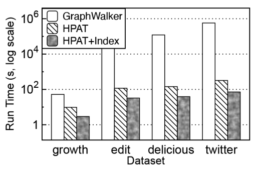


Fig. 11. Breakdown.

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

6.3 Piecewise Breakdown

In this section, we analyze the impacts of two critical optimizations: HPAT sampling and auxiliary index (Section 3.4). We utilize temporal node2vec as the example application for this analysis and execute the study in an in-memory environment. The baseline for comparison is GraphWalker. It is important to note that the lock-free parallel construction of the *TEA+* data structures, including searching for candidate edge sets and PAT/HPAT construction, also incurs some time overhead, albeit minimal. For a detailed evaluation of this aspect, please refer to our preliminary work [16].

HPAT Sampling Optimization. As depicted in Figure 11, on average, our HPAT optimization achieves an impressive speedup of 812.55× compared to the baseline. The most substantial speedup is observed in the twitter dataset, where *TEA+* attains an astounding 1,788× speedup. Even the smallest speedup is quite significant at 5.4×. Through analysis, we discern that the primary factor contributing to the variation in speedup is the disparity in the average degrees across different graph datasets. Specifically, since the time complexity of the baseline is $O(D)$ and that of *TEA+* is $O(log(log(D)) + log(D))$, our sampling process is relatively indifferent to the degree D

Auxiliary Index Optimization. This contributes an additional speedup ranging from 2.75 to 3.45× to *TEA+*. It is imperative to recall that auxiliary index optimization aims to expedite the location of the alias table trunks relevant to our analysis. This enhancement becomes increasingly significant as the HPAT optimization drastically reduces the sampling time. In line with the first optimization, the largest gain is observed for the twitter dataset, with a reduction in time from 320.1 seconds to 92.93 seconds, whereas the smallest speedup is observed for the growth dataset, with a reduction in time from 9.66 seconds to 3.52 seconds. The speedup is attributed to the reduction in time complexity from $O(log(log(D))+log(D))$ to $O(log(log(D)))$. It is worth noting that the impact of the degree on this optimization is analogous to that of the HPAT optimization across different datasets.

6.4 Evaluating *TEA+*'s Efficacy on Non-Power-Law Graphs

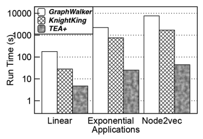
TEA+ enjoys less speedups on non-power-law graphs compared to power-law graphs. However, *TEA+* can still get decent speedups on non-power-law graphs. As highlighted in PowerGraph [12], natural graphs typically found in real-world scenarios exhibit highly skewed power-law degree distributions. In cases of non-power-law graphs, such as random graphs, the impact of *TEA+* is less pronounced. The reason is that such non-power-law graphs often hold more vertices with smaller degrees than power-law graphs.

First, we briefly describe two background points. Power-law graphs follow the degree distribution rule of $P(d) \sim d^{-\alpha}$, where α is the power-law exponent which is usually in the range of 2 to 3, and $P(d)$ is the probability of a vertex having d connections (degree d) [4]. Additionally, the R-MAT (Recursive Matrix) model can generate synthetic graphs that mimic the properties of real-world networks. These properties include power-law degree distribution, small-world property, and self-similarity [5]. One can change α to make R-MAT generate non-power-law graphs.

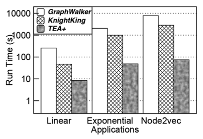
Second, we deliberately adjust the R-MAT parameters to create two RMAT graphs that do not exhibit a power-law degree distribution in Table 1. This is achieved by carefully controlling the parameters. Additionally, we employ linear regression to calculate the power-law exponent of the generated graphs, ensuring that the exponent falls outside the typical range of $[2, 3]$.

The first RMAT graph, denoted as *RMAT1*, consists of 16.777 million vertices and 536.87 million edges, generated with parameters {0.5, 0.1, 0.1, 0.3}. RMAT1 exhibits a maximum degree of 2,594, and its power-law exponent is 3.0602. The second graph, *RMAT2*, comprises 67.108 million vertices and 1.073 billion edges, created with parameters {0.45, 0.15, 0.15, 0.25}, and has a maximum degree of 1,827. Its power-law exponent is 3.5265.

As depicted in Figure 12, *TEA+* still can achieve a significant speedup over other methods on both datasets, ranging from 4.9× to 171.3× on the RMAT1 dataset, and from 3.5× to 104.2× on the RMAT2 dataset. *TEA+* shows a somewhat reduced speedup on both RMAT graphs compared to power-law graphs, where it achieves speedups of up to three orders of magnitude. This outcome largely stems from *TEA+*'s key advantage: its improved sampling time complexity, reduced from $O(D)$ to $O(\log(\log(D)))$. This improvement is linked to vertex degree numbers. Additionally, as

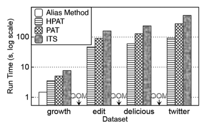


(a) RMAT1.

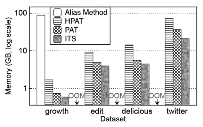


(b) RMAT2.

Fig. 12. Evaluation on RMAT graphs.



(a) Runtime comparison.



(b) Memory usage.

Fig. 13. Sampling methods comparison.

random walks more frequently visit higher-degree vertices (as observed in KnightKing [42]), this aspect significantly influences performance. Thus, our RMAT graphs, with their lower degree variance and non-power-law distribution, show less improvement in sampling time complexity. This is especially evident when contrasted with power-law graphs, which have higher degree variance. Nonetheless, *TEA+* still achieves considerable speedup, up to 171.3× over baseline methods, even on non-power-law graphs.

6.5 Comparison of Various Sampling Methods

This section evaluates the performance implications of employing HPAT and PAT compared to the current Monte Carlo sampling methods, namely ITS [27] and the alias method [23]. Notably, ITS can be directly integrated into *TEA+* since the sampling space is organized in a time-decreasing order, which is conducive to the construction of the ITS.

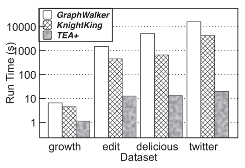
Figures 13(a) and 13(b) illustrate the runtime and memory usage comparisons for temporal node2vec, respectively. The alias method exhibits the shortest runtime on the growth dataset; however, it is unable to handle other datasets due to its excessive space consumption (refer to Section 3.1). Even within the growth dataset, the alias method is only marginally faster than HPAT, at 1.38×, but at the cost of a staggering 51.7× increase in memory usage compared to HPAT. In the case of other datasets, HPAT emerges as the fastest sampling method, followed closely by PAT. More specifically, HPAT achieves a speedup ranging from 1.43× to 2.97× compared to PAT, and PAT achieves a speedup ranging from 1.22 to 1.89× compared to ITS. In terms of memory usage, PAT and ITS have similar memory footprints, with PAT consuming, on average, only 1.26× more space than ITS. Furthermore, HPAT consumes, on average, 1.95× more space than PAT.

6.6 Expanded Evaluation of Additional Random Walk Applications

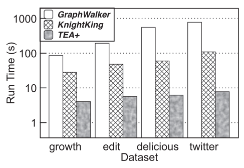
To comprehensively assess the efficiency of *TEA+* across various applications, we have also chosen four additional, widely used random walk applications as representative examples for further evaluation. The additional applications are shown in the following:

- *Random Walk Domination* [25]: We initiate a random walk of length 6 starting from each vertex in the graph. The objective is to identify a set of vertices that exhibit the maximum influence diffusion across the network.
- *Graphlet Concentration (Graphlet)* [31]: Focusing on triangles as a case study, we initiate 100,000 random walks, each of length 4, to estimate the proportion of triangles within the graph.
- *Personalized PageRank* [10]: To approximate the **Personalized PageRank (PPR)** values, we conduct 2,000 random walks of length 10 from each query source vertex. This approach has been demonstrated to be sufficient for ensuring accuracy.
- *SimRank* [18]: To compute the expected meeting time, we initiate 2,000 random walks, each of length 11, from each vertex in the query pair.

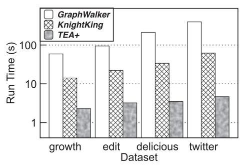
In these four applications, as depicted in Figure 14, *TEA+* consistently demonstrates strong performance. For **Random Walk Domination (RWD)**, which resembles the exponential weight



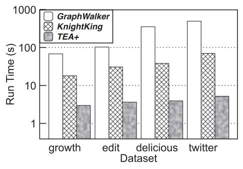
(a) RWD.



(b) Graphlet.



(c) PPR.



(d) SR.

Fig. 14. Comparison of *TEA+* with GraphWalker and KnightKing across various applications.

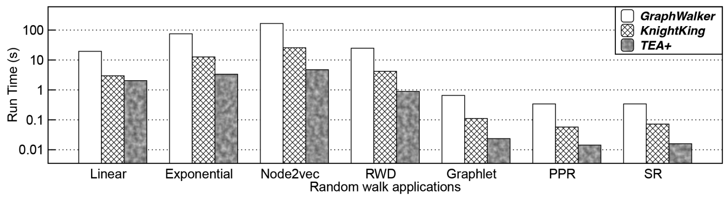


Fig. 15. Evaluation on the industry dataset.

random walk but with a shorter walk length, *TEA+* achieves a speedup ranging from 3.94× to 798.48× compared to baseline methods. Applications like Graphlet, PPR, and **SimRank (SR)** involve random walks with fixed numbers and lengths across all datasets. Consequently, the runtime for these applications is relatively consistent across different datasets, allowing *TEA+* to attain a speedup of up to 101.05×.

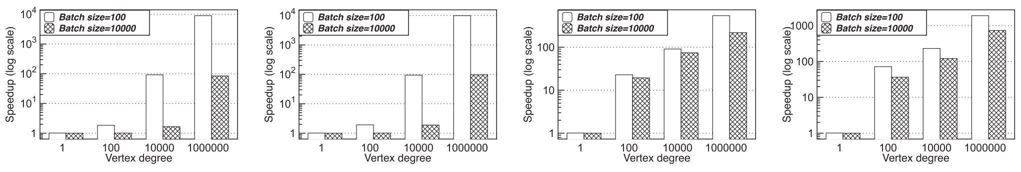
6.7 Evaluation on Industry Datasets

To study the practical impacts of *TEA+*, we evaluate our system on an Ant Group dataset (https://www.atecup.cn/dataSetDetailOpen/2) regarding financial risk prediction. This dataset comprises 5.3 million vertices and 30.8 million edges, with the maximum degree number reaching 10,628, serving as a substantial input for our analysis. Each vertex represents a user, and each edge represents a trade between two users. We then applied seven widely used random walk applications to the financial network to mine its structural information, including Linear Temporal Weight Random Walk, Exponential Temporal Weight Random Walk, Temporal Node2vec, RWD, Graphlet Concentration (Graphlet), PPR, and SR. It is important to note that RWD, Graphlet, PPR, and SR all utilize exponential temporal weight random walks.

Figure 15 shows that *TEA+* can achieve a significant speedup, ranging from 1.45× to 35.02×, across seven widely used random walk applications. This performance is consistent with the datasets mentioned in Table 1, where *TEA+* also demonstrates good efficiency. The acceleration is primarily attributed to the reduction in sampling time complexity, which improves from $O(\log(D))$ to $O(\log(\log(D)))$ for Linear Temporal Weight Random Walk, and from $O(D)$ to $O(\log(\log(D)))$ for other applications. However, due to the relatively modest size of this dataset and its maximum degree number being significantly lower than those found in other datasets (i.e., twitter's maximum degree number ascends to millions, in contrast to this dataset's maximum of merely 10.6K), the observed speedup on this dataset is not as remarkable as that witnessed with other larger datasets, where *TEA+* can achieve speedups of three orders of magnitude on the twitter dataset. Even so, *TEA+* can still get good performance on the industry dataset.

6.8 Performance Evaluation of Graph Modifications

To conduct a more comprehensive analysis of the graph modification, we evaluate edge insertion, edge deletion, vertex insertion, and vertex deletion separately.



(a) Edge insertion. (b) Edge deletion. (c) Vertex insertion. (d) Vertex deletion.

Fig. 16. Evaluation of graph modifications.

Figure 16(a) illustrates the speedup of our edge insertion strategy compared to the naive baseline, which rebuilds the HPAT from scratch. This speedup is influenced by two main factors: the batch size of incoming edges and the vertex degree size. The latter is particularly impactful as it determines the workload difference between our method and the naive approach. In our evaluation, we consider batch sizes of 100 and 10k for vertex degrees of 1, 100, 10k, and 1 million. Generally, the speedup approaches 1 when the vertex degree is much smaller than the batch size. For batch sizes equal to the vertex degree, the speedup reaches 1.82× and 1.65× for batch sizes of 100 and 10k, respectively. However, when the degree size significantly exceeds the batch size, the speedup becomes substantial. Notably, for a degree size of 10^6, a batch size of 100 achieves a speedup of 8,975×, whereas a batch size of 10k results in a 79.3× speedup.

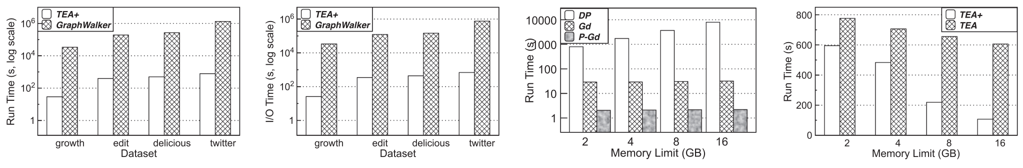
Figure 16(b) illustrates the speedup achieved by our incremental edge deletion strategy compared to the baseline. It is evident that for edge deletion, our strategy attains a higher speedup than edge insertion. This is attributed to the fact that the edge deletion process does not necessitate the construction of HPATs, whereas edge insertion requires building HPATs for the newly incoming edges. Edge deletion merely involves the removal of HPATs and the liberation of the corresponding memory. Specifically, with batch sizes of 100 and 10k, our project achieves a speedup of up to 9,853× and 96.4×, respectively, when the vertex degree is 10^6.

Figure 16(c) illustrates the speedup achieved by our incremental vertex insertion strategy compared to the baseline method, which rebuilds the sampling index for the affected vertices. The affected vertices encompass both the newly added vertex and those connected to the added edges, as the insertion of vertices can be conceptualized as the addition of a batch of edges. It is evident that *TEA+* achieves a significant speedup, up to two orders of magnitude, over the baseline. Notably, the speedup becomes more pronounced when the inserted vertices have higher degrees, resulting in a greater number of added edges and, consequently, more affected vertices. This trend underscores the efficiency of *TEA+* in handling vertex insertions in large-degree graphs.

Figure 16(d) illustrates the speedup achieved by our incremental vertex deletion strategy compared to the baseline, which involves rebuilding the sampling index for vertices affected by the deletion. Upon the deletion of a vertex, a batch of edges is consequently removed, necessitating the reconstruction of the index for vertices connected to these edges. It is observed that *TEA+* can achieve up to 1,890.6× speedup compared to the rebuild method. This efficiency is due to *TEA+*'s ability to selectively delete only the affected trunks, rather than rebuilding all trunks, thereby significantly enhancing the process of handling vertex deletions.

6.9 Studying *TEA+* in an Out-of-Core Setting

Figure 17(a) shows the runtime comparison of *TEA+* and GraphWalker in an out-of-core environment, using the temporal node2vec application. In this mode, *TEA+* stores PATs on disk for all vertices. On average, *TEA+* is 713× faster than GraphWalker, with peak and minimum speedups of 1, 172× on the twitter dataset and 115× on the growth dataset, respectively. Considering the significant role of disk I/O in this setting, Figure 17(b) further analyzes the I/O performance. The



(a) Overall performance. (b) Disk I/O comparison. (c) Index selection. (d) Memory sensitivity.

Fig. 17. Out-of-core execution analysis.

average, highest, and lowest I/O speedups achieved are 480.4×, 1107.8× (on the twitter dataset), and 130.3× (on the growth dataset), respectively.

Disk I/O Comparison. In the out-of-core execution environment, disk I/O is the primary contributor to runtime. Since *TEA+* utilizes the prefix-sum of edge trunks to select the relevant trunk for loading, its I/O complexity is $O(\text{trunkSize})$ as discussed in Section 3.2. In contrast, GraphWalker must load D neighbors into memory for sampling, resulting in both sampling and I/O complexities of $O(D)$. Given that disk I/O operates at a significantly slower speed compared to CPUs, the computation involved in sampling consumes considerably less time than loading neighbors from the disk. This accounts for the consistent trends observed in Figure 17(a) and (b).

Index Selection. Figure 17(c) showcases the efficacy of different index selection algorithms for allocating PATs or HPATs between memory and disk, as discussed in Section 4. These algorithms include dynamic programming (*DP*), the greedy algorithm (*Gd*), and the parallel greedy algorithm (*P-Gd*) utilizing 16 threads. The performance of DP is notably influenced by memory size, with its time complexity being $O(|V|M)$, where V is the vertex set and M is the memory size. In contrast, Gd and P-Gd, with time complexities of $O(|V|\log(|V|))$, are less affected by memory size. Gd outperforms DP with a speedup of up to 246.8×, benefiting from the reduced time complexity from $O(|V|M)$ to $O(|V|\log(|V|))$. Furthermore, P-Gd leverages multi-threading to achieve a speedup of up to $3,574\times$ compared to DP, showcasing significant scalability.

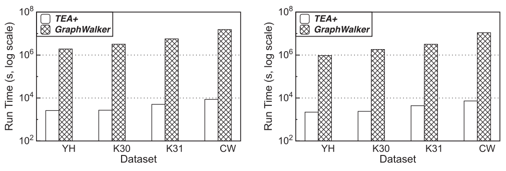
Memory Size Sensitivity. To show the efficiency of our degree-aware hybrid storage architecture, we compare *TEA+* with the previous version *TEA* [16] on the twitter dataset under different memory limit. The memory limits we choose are 16, 8, 4, and 2 GB. *TEA* only adjusts the trunk size to fit the memory size, whereas *TEA+* uses the degree-aware hybrid storage architecture. The evaluations shows that *TEA+* can get better performance on larger memory limits, which can get $1.3 \sim 5.6\times$ more speedup than *TEA*. This is because on the larger memory size, *TEA+* will have more space to store the HPATs and PATs in memory to reduce the disk I/O and improve performance.

6.10 Performance Evaluation on Large Datasets

This evaluation uses four large graph datasets: YahooWeb (*YH*) [1] with 1.4 billion vertices and 6.6 billion edges (108 GB), Kron30 (*K30*) with 1 billion vertices and 32 billion edges (638 GB), Kron31 (*K31*) with 2 billion vertices and 64 billion edges (1.4 TB), both generated by the Graph500 Kronecker model (https://graph500.org/), and the largest, CrawlWeb (*CW*) [2], with 3.5 billion vertices and 128 billion edges (2.6 TB). K30 and K31 are generated using the Graph500 Kronecker model.

Evaluation Methodology. To manage the long runtimes of baseline methods, which can exceed thousands of hours, we used an estimation technique from random walk engines like KnightKing [42] for their performance evaluation. This involves running baselines on a small, random sample of walkers (1%–6%), and using linear regression to estimate total execution time, based on the linear scaling relationship between computation time and walker numbers. We have validated this method through full runs in some cases, confirming that our estimates deviate less than 2% from actual runtimes.

Figure 18 demonstrates *TEA+*'s performance on large datasets, focusing on overall efficiency and disk I/O time. By incorporating PAT, *TEA+* significantly enhances performance, achieving speedups from 731.6× to 1,781.8×. In terms of disk I/O, the speedups range from 648.2× to 1,506.1×. The key to *TEA+*'s speedup, particularly in out-of-core mode, is its selective memory loading using PAT. Instead of loading the entire sampling index, it loads only the relevant trunk into memory, requiring $O(trunkSize)$ disk I/O, far less than the $O(D)$ needed for the full alias method. This not only minimizes disk I/O but also speeds up the sampling process.



(a) Overall performance. (b) Disk I/O time.

Fig. 18. Out-of-core evaluation on large datasets.

7 RELATED WORK

Recent random walk applications focus on static graph engines like DrunkardMob [22] and GraphWalker [37], which operate on single machines, with the former loading datasets into memory and the latter improving data handling and walk updates. KnightKing [42] is a distributed engine optimizing rejection sampling for static graphs, whereas FlashMob [41] and ThunderRW [35] address irregular memory access. NosWalker [38] utilizes a pre-sampling strategy, caching pre-sampled vertices in memory to accelerate out-of-core random walks. However, this approach becomes inefficient as the pre-sampled vertices turn invalid when candidate edge sets change. These engines struggle with temporal graphs due to high time and space complexities.

Random walks in temporal graphs play a critical role in graph embedding techniques [24, 43]. It is important to emphasize that conventional static random walk models are not suited for temporal graphs due to their inability to accommodate the temporal dimension. In the context of random walk models for temporal spaces, CTDNE [28] introduces an exponential weight random walk, which has gained widespread adoption in temporal graph random walks [17, 39]. CAW [39] introduces *Causal Anonymous Walks* for inductive representation learning in temporal graphs which employs a distinct random walk generation strategy. EHNA [17] presents a temporal adaptation of node2vec by integrating a temporal random walk algorithm within a stacked LSTM architecture. While this method achieves high accuracy, it poses challenges in terms of programming complexity and exhibits slower execution efficiency compared to CTDNE.

8 CONCLUSION

This article presented *TEA+*, a unique random walk engine tailored for temporal graphs, capable of handling dynamic changes like edge and vertex insertions and deletions. We also introduced the PAT and HPAT strategies with an auxiliary index to enhance the sampling speed. *TEA+* offers a user-friendly interface for developing various temporal random walk algorithms. Additionally, its degree-aware hybrid storage design optimizes performance for different memory sizes, achieving up to a thousandfold performance improvement over current random walk engines.

REFERENCES

[1] Yahoo Webscope. n.d. Home Page. Retrieved March 19, 2024 from http://webscope.sandbox.yahoo.com
[2] Web Data Commons. n.d. The 2012 Common Crawl Graph. Retrieved March 19, 2024 from http://webdatacommons.org
[3] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael B. Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph analytics on evolving data. In *Proceedings of ASPLOS*. 133–145.
[4] Anna D. Broido and Aaron Clauset. 2019. Scale-free networks are rare. *Nature Communications* 10, 1 (2019), 1017.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of ICDM*. 442–446.

[6] 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 *Proceedings of EuroSys*. 85–98.

[7] 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 SIGKDD*. 257–266.

[8] Michael Cochez, Petar Ristoski, Simone Paolo Ponzetto, and Heiko Paulheim. 2017. Biased graph walks for RDF graph embeddings. In *Proceedings of WIMS*. Article 21, 12 pages.

[9] Peng Cui, Xiao Wang, Jian Pei, and Wenwu Zhu. 2019. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2019), 833–852.

[10] 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.

[11] Tao-Yang Fu, Wang-Chien Lee, and Zhen Lei. 2017. HIN2Vec: Explore meta-paths in heterogeneous information networks for representation learning. In *Proceedings of CIKM*. 1797–1806.

[12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of OSDI*. 17–30.

[13] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable feature learning for networks. In *Proceedings of SIGKDD*. 855–864.

[14] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of NIPS*. 1024–1034.

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

[16] Chengying Huan, Shuaiwen Leon Song, Santosh Pandey, Hang Liu, Yongchao Liu, Baptiste Lepers, Changhua He, Kang Chen, Jinlei Jiang, and Yongwei Wu. 2023. Tea: A general-purpose temporal graph random walk engine. In *Proceedings of EuroSys*. 182–198.

[17] Shixun Huang, Zhifeng Bao, Guoliang Li, Yanghao Zhou, and J. Shane Culpepper. 2020. Temporal network representation learning via historical neighborhoods aggregation. In *Proceedings of ICDE*. IEEE, 1117–1128.

[18] Glen Jeh and Jennifer Widom. 2002. SimRank: A measure of structural-context similarity. In *Proceedings of SIGKDD*. 538–543.

[19] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: Generalized incremental graph processing via graph triangle inequality. In *Proceedings of EuroSys*. 17–32.

[20] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of SIGKDD*. 1269–1278.

[21] Jérôme Kunegis. 2013. KONECT: The Koblenz network collection. In *Proceedings of WWW*. 1343–1350.

[22] Aapo Kyrola. 2013. DrunkardMob: Billions of random walks on just a PC. In *Proceedings of RecSys*. 257–264.

[23] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. 2014. Reducing the sampling complexity of topic models. In *Proceedings of SIGKDD*. 891–900.

[24] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. 2018. Adaptive graph convolutional neural networks. In *Proceedings of AAAI*. 3546–3553.

[25] Rong-Hua Li, Jeffrey Xu Yu, Xin Huang, and Hong Cheng. 2014. Random-walk domination in large graphs. In *Proceedings of ICDE*. 736–747.

[26] 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 SIGMOD*. ACM, 135–146.

[27] Frederic P. Miller, Agnes F. Vandome, and John McBrewster (Eds.). 2010. *Inverse Transform Sampling*. VDM Publishing.

[28] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-time dynamic network embeddings. In *Proceedings of WWW*. 969–976.

[29] 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 SC*. IEEE, 56.

[30] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. In *Proceedings of SIGKDD*. 701–710.

[31] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* 23, 2 (2007), e177–e183.

[32] 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 SIGMOD*. 1797–1812.

[33] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of SIGMOD*. 417–430.

[34] Guolei Sun and Xiangliang Zhang. 2017. Graph embedding with rich information through bipartite heterogeneous network. *CoRR abs/1710.06879* (2017).

[35] Shixuan Sun, Yuhang Chen, Shengliang Lu, Bingsheng He, and Yuchen Li. 2021. ThunderRW: An in-memory graph random walk engine. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1992–2005.

[36] John von Neumann. 1951. Various techniques used in connection with random digits. In *Monte Carlo Method.* National Bureau of Standards Applied Mathematics Series, Vol. 12. National Bureau of Standards, 36–38.

[37] 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 *Proceedings of ATC*. 559–571.

[38] Shuke Wang, Mingxing Zhang, Ke Yang, Kang Chen, Shaonan Ma, Jinlei Jiang, and Yongwei Wu. 2023. NosWalker: A decoupled architecture for out-of-core random walk processing. In *Proceedings of ASPLOS*. 466–482.

[39] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive representation learning in temporal networks via causal anonymous walks. In *Proceedings of ICLR*.

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

[41] Ke Yang, Xiaosong Ma, Saravanan Thirumuruganathan, Kang Chen, and Yongwei Wu. 2021. Random walks on huge graphs at cache efficiency. In *Proceedings of SOSP*. 311–326.

[42] 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 SOSP*. 524–537.

[43] 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 SIGKDD*. 974–983.

[44] 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 SIGKDD*. 2672–2681.

[45] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph sampling based inductive learning method. In *Proceedings of ICLR*.

[46] Ziqian Zeng, Xin Liu, and Yangqiu Song. 2018. Biased random walk based social regularization for word embeddings. In *Proceedings of IJCAI*. 4560–4566.

[47] Yifeng Zhao, Xiangwei Wang, Hongxia Yang, Le Song, and Jie Tang. 2019. Large scale evolving graphs with burst detection. In *Proceedings of IJCAI*. 4412–4418.

[48] Dongyan Zhou, Songjie Niu, and Shimin Chen. 2018. Efficient graph computation for Node2Vec. *CoRR abs/1805.00280* (2018).

[49] 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. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.

[50] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of OSDI*. 301–316.

[51] Yuan Zuo, Guannan Liu, Hao Lin, Jia Guo, Xiaoqian Hu, and Junjie Wu. 2018. Embedding temporal network via neighborhood formation. In *Proceedings of SIGKDD*. 2857–2866.

Received 4 July 2023; revised 26 February 2024; accepted 1 March 2024