From random-walks to graph-sprints: a low-latency node embedding framework on continuous-time dynamic graphs

Ahmad Naser Eddin*
Feedzai
Portugal
ahmad.eddin@feedzai.com

Jacopo Bono*
Feedzai
Portugal
jacopo.bono@feedzai.com

David Aparício
DCC/FCUP, University of Porto
Portugal
daparicio@dcc.fc.up.pt

Hugo Ferreira
Feedzai
Portugal
hugo.ferreira@feedzai.com

João Ascensão
Feedzai
Portugal
joao.ascensao@feedzai.com

Pedro Ribeiro
DCC/FCUP, University of Porto
Portugal
pribeiro@dcc.fc.up.pt

Pedro Bizarro
Feedzai
Portugal
pedro.bizarro@feedzai.com

ABSTRACT

Many real-world datasets have an underlying dynamic graph structure, where entities and their interactions evolve over time. Machine learning models should consider these dynamics in order to harness their full potential in downstream tasks. Previous approaches for graph representation learning have focused on either sampling k-hop neighborhoods, akin to breadth-first search, or random walks, akin to depth-first search. However, these methods are computationally expensive and unsuitable for real-time, low-latency inference on dynamic graphs. To overcome these limitations, we propose *graph-sprints* a general purpose feature extraction framework for continuous-time-dynamic-graphs (CTDGs) that has low latency and is competitive with state-of-the-art, higher latency models. To achieve this, a streaming, low latency approximation to the random-walk based features is proposed. In our framework, time-aware node embeddings summarizing multi-hop information are computed using only single-hop operations on the incoming edges. We evaluate our proposed approach on three open-source datasets and two in-house datasets, and compare with three state-of-the-art algorithms (TGN-attn, TGN-ID, Jodie). We demonstrate that our *graph-sprints* features, combined with a machine learning classifier, achieve competitive performance (outperforming all baselines for the node classification tasks in five datasets). Simultaneously, *graph-sprints* significantly reduce inference latencies, achieving up to 9 times faster inference in our experimental setting.

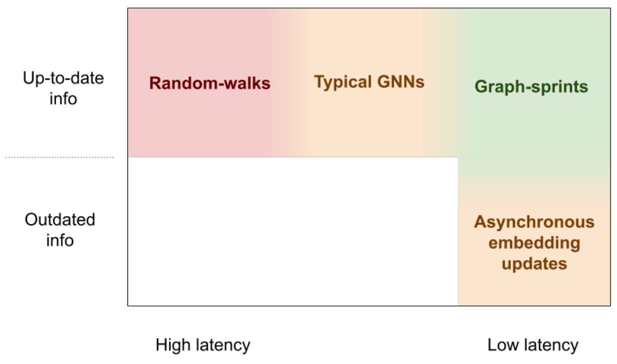

Figure 1: CTDGs Approaches Overview. Current approaches either compute embeddings using real-time information but sacrificing latency, or compute low-latency embeddings but use outdated information in the computation. Our proposed method, *graph-sprints*, computes embeddings in low latency using real-time information.

1 INTRODUCTION

Many real-world datasets have an underlying graph structure. In other words, they are characterized not only by their individual data points but also by the *relationships* between them. Moreover, they are typically dynamic in nature, meaning that the entities and their interactions change over time. Examples of such systems are social networks, financial datasets, and biological systems [4, 26, 34]. Dealing with dynamic graphs is more challenging compared to static graphs, especially if the graphs evolve in continuous time (also known as continuous-time dynamic graphs or CTDGs). The majority of machine learning models on graph datasets are based on graph neural networks (GNNs), achieving state-of-the-art performance [31]. A few deep neural network architectures emerged to deal with CTDGs in the past years [8, 10, 13, 15, 20, 23, 32]. One drawback of these approaches is that one either needs to sample k-hop neighborhoods to compute the embeddings (e.g. [20]) or perform random-walks (e.g. [13]). Both cases are computationally costly, resulting in high inference latencies. One solution is to decouple the inference from the expensive graph computations, like in APAN [28]. Performing the graph aggregations asynchronously results in inference using outdated information (Figure 1). However, in large data and high-frequency use-cases, such as detecting fraud in financial transactions, one requires low-latency solutions

*These authors contributed equally to this work

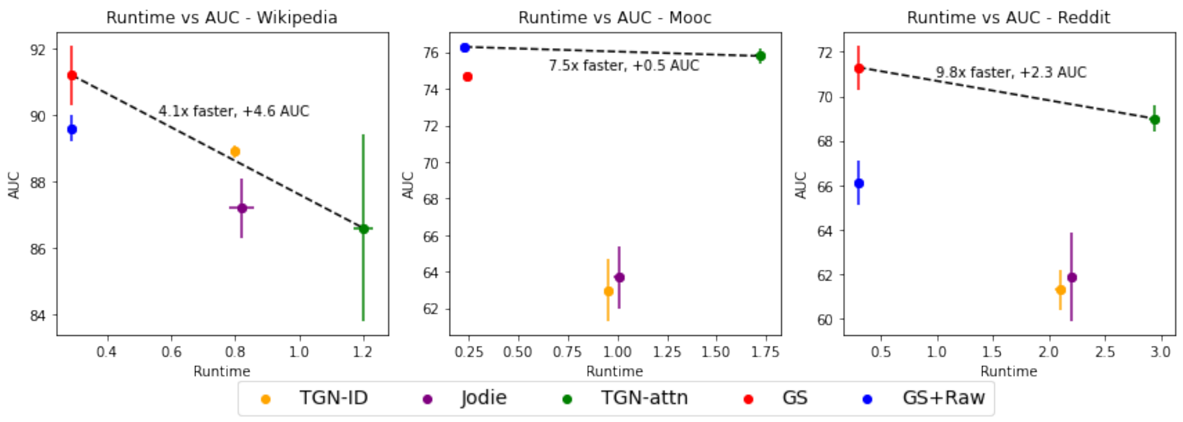

Figure 2: AUC vs. Runtime trade-off. Our proposed methods based on *graph-sprints* (GS or GS-Raw), allow for low latency inference while outperforming state-of-the-art methods in terms of AUC on node classification tasks. X-axis represents time in seconds to process 200 batches of size 200, Y-axis represents test AUC. Error bars denote the standard deviation over 10 random seeds.

that preferably use the most up-to date information to increase detection.

In this work, we propose a versatile graph feature extraction framework, which computes node embeddings in the form of features that characterize a node's neighborhood in dynamic graphs. Importantly, our proposed framework is designed for low-latency settings while still using the most up-to-date information during the embedding calculations. Since we derive our framework starting from random-walk based methods, we term our method *graph-sprints*. The learned *graph-sprints* features can then be used in any downstream system, for example in a machine learning model or a rule-based system. We show how the proposed *graph-sprints* features, combined with a neural network classifier, are faster to run while not sacrificing in predictive performance compared with the higher-latency GNNs.

The remainder of the paper is organized as follows. We first discuss the proposed *graph-sprints* framework in Section 2.2. In Section 3, we evaluate our framework using three open-source datasets from different domains, and two in-house datasets from the money laundering domain. We discuss related work in Section 4 and in Section 5 we put forward our main conclusions.

2 METHODS

2.1 Random-walk based features

Before describing our *graph-sprints* framework, we briefly summarize a random-walk based feature extraction framework. Our aim is to propose a set of general-purpose features that characterize a node's neighborhood, and for which we will derive efficient *graph-sprints* computations in the next section. The random-walk based feature extraction framework requires the following steps, to generate a node embedding for target seed node.

(1) **Select the seed node.** This selection depends on the use-case, and for CTDGs typically one considers entities involved in new activity, for instance if the change on the graph is adding a new edge between two nodes, then each of these two nodes could be a candidate for a seed node.
(2) **Perform random-walks starting from the seed nodes.** During the random-walks, relevant data such as node or edge features of the traversed path are collected. The type of random-walks influences what neighborhood is summarized in the extracted features. Walks can be (un)directed, biased, and/or temporal.
(3) **Summarize collected data.** The data collected during the random-walks is aggregated into a fixed set of features, characterizing each seed node's neighborhood. Examples of such aggregations are the average of encountered numerical node or edge features, the maximum of encountered out-degree, counts of a category for categorical node or edge features, etc.

The computation of these features is costly, because multiple random-walks need to be generated for each seed node. For CTDGs, one would have to compute such features each time an edge arrives. This is infeasible for high-frequency use-cases such as fraud detection in financial transactions, where a low-latency decision is needed. In the next section, we derive an efficient approximation to the above random-walk based features.

2.2 Graph-sprints: streaming graph features

In this section, we propose approximations to random-walk based features described in Section 2.1. Our aim in this section is to optimize the computation of such features by exploiting recurrence and abolishing the need to execute full random-walks (Figure 3).

2.2.1 *Assumptions.*

For our approximations to be reliable, we make the following assumptions: the input graph is a CTDG with directed edges, edges have timestamps and the temporal walks respect time, in the sense that the next explored edge is older than the current edge. With these assumptions, one can unfold any directed temporal walk as a time-series (Figure 3A and B).

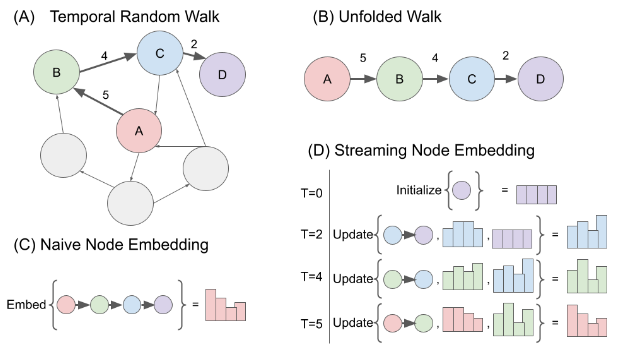

Figure 3: From random-walks to graph-sprints. Edges have a timestamp feature (numbers) representing the time that a relationship was created. (A) A temporal random-walk is traversed from the most recent interaction A-B towards older interactions. (B) The same random-walk can be seen as a time-series of edges. (C) Based on a full temporal random-walk, one can compute embeddings by aggregating encountered feature values (section 2.1). (D) One can compute similar embeddings in a streaming setting, from only the new edge and the existing embeddings of the involved nodes.

2.2.2 Streaming histograms as node embeddings. Given the above assumptions, we now formalize the approximation of random-walk based aggregations described in section 2.1.

In this framework, we do not consider random-walks with a fixed number of hops, and instead consider infinite walks. The importance of older information compared to newer is controlled by a factor α. A larger α gives more weight to features further away in the walk, and we can therefore consider α the parameter that replaces the number of hops. Formally, let $\vec{s}_i$ be a histogram with L bins, represented as an L-dimensional vector and characterizing the distribution of a feature f in the neighborhood of node i. A full infinite walk starting at node 0 computes the histogram $\vec{s}_0$ as:

$$\vec{s}_0 = \sum_{i=0}^{\infty} \alpha^i (1-\alpha)\vec{\delta}(f_i) \tag{1}$$

where α is a discount factor between 0 and 1, controlling the importance of distant information in the summary $\vec{s}_0$, and i denotes the hops of the walk ($i = 0$ being the newest node, or in other words the seed node of the infinite walk). f_i is the feature value at node i and $\vec{\delta}(f_i)$ is an L-dimensional vector with element $\vec{\delta}_j = 1$ if the feature value f_i falls within bin j and $\vec{\delta}_j = 0$ for all other elements. Equation 1 then implements a streaming counts per bin, where older information is gradually forgotten. If the feature f_i is a node feature, then the value is taken from the current node. If it is an edge feature, then the feature value is taken from the edge connecting the current node and the chosen neighbor.

One could compute multiple such summaries per node, one for each node or edge feature of interest, and together they would summarize a neighborhood. The key idea is that we can now approximate the infinite random-walks, i.e., the infinite sum of equation 1, by performing only a finite number of $k \geq 1$ hops, followed by choosing a random neighbor of the last encountered node and choosing an available summary $\vec{s}_k$ of that neighbor randomly, where $\vec{s}_k$ is defined as

$$\vec{s}_k = \sum_{i=0}^{\infty} \alpha^i (1-\alpha)\vec{\delta}(f_{i+k}) \tag{2}$$

With this strategy, we can approximate the summary $\vec{s}_0$ from equation 1 recurrently using

$$\vec{s}_0 \approx \sum_{i=0}^{k-1} \alpha^i (1-\alpha)\vec{\delta}(f_i) + \alpha^k \vec{s}_k \tag{3}$$

Compared with equation 1, one now truncates the sum after k terms. Note that whenever the last histogram $\vec{s}_k$ is normalized such that the bins sum to 1, e.g. using a uniform initialization for terminal nodes, equation 3 guarantees that all subsequent histograms will be normalized in the same way. Since we are interested in low-latency methods, we take the limit of $k = 1$ and Equation 3 becomes a streaming histogram:

$$\vec{s}_0 \leftarrow (1-\alpha)\vec{\delta}(f_0) + \alpha\vec{s}_1 \tag{4}$$

The hyperparameter α can be chosen to depend on the number of hops or on time. When discounting by hops, this discount factor α is a fixed number between 0 and 1. When discounting by time, the factor is made dependent on the difference in edge timestamps, for example exponentially as illustrated in Equation 5 or hyperbolically as in Equation 6.

$$\alpha(t_k, t_{k-1}) = \exp(-|t_k - t_{k-1}|/\tau) \tag{5}$$

$$\alpha(t_k, t_{k-1}) = \frac{1}{1 + |t_k - t_{k-1}|/\tau} \tag{6}$$

Where τ is a chosen timescale and t_k is the timestamp of edge at hop k. Similarly, the resulting α will be a value between 0 and 1, and the bigger the time difference between the two edges the smaller α will be. Therefore, giving more importance to more recent connections.

Using equation 4, one could approximate N (biased) random-walks by sampling N neighbors (non-uniformly), and subsequently combining the resulting histograms e.g. by averaging. This would require performing N 1-hop look-ups each time.

Instead of that, we can increase efficiency even further by removing any stochasticity and updating a node's histogram at each edge arrival, combining the histograms of the two nodes involved in the arriving edge, as shown in equation 7:

$$\vec{s}_0 \leftarrow \beta\vec{s}_0 + (1-\beta)\left((1-\alpha)\vec{\delta}(f_0) + \alpha\vec{s}_1\right) \tag{7}$$

In this way we combine all neighbours' information implicitly using a moving average over time.

Hyperparameter β is another discount factor between 0 and 1, controlling how much to focus on recent neighbor information in contrast to older information and which can take similar dependencies on time as equations 5 and 6. In this way, we can update histograms in a fully streaming setting, using only information of each arriving edge. We term this procedure *graph-sprints* and summarize it in algorithm 1.

Compared to equation 4, one can observe that the remaining sampling over single-hop neighbors is abolished, at the cost of imposing a more strict dependence on time. The advantage of algorithm 1 is that no list of neighbors needs to be stored. Moreover, algorithm 1 can be applied in parallel to both the source node and the destination node, and therefore edges are not required to be directed. In fact, while we derived equation 7 from random-walks, the attentive reader can notice that it can be interpreted as a special case of message passing where all neighbor summaries are aggregated using a weighted average, with weights that are biased by recency, and where the average is computed in a streaming fashion over time.

One special type of feature are the degree features (in- and out-degree). To avoid accumulating degrees over time, we propose to implement a streaming count of degrees per node. Every time an edge involving node u arrives, we compute

$$d_u = d_u \exp\left(-\Delta t / \tau_d\right) + 1 \tag{8}$$

where d_u denotes either in- or out-degree of node u, Δt denotes the time differences between the current edge involving node u and the previous one, and τ_d is a timescale for the streaming counts.

Algorithm 1 Graph-sprints (eq. 7)

```
Require: $EdgeStream$                    ▷ Stream of arriving edges $e_{i,j}$
Require: $\mathcal{F}$                   ▷ Set of features for GS (e.g., node degree)

  for $e_{v,u} \in EdgeStream$ do
      Get $\vec{s}_u, \vec{s}_v$                  ▷ Current summaries of nodes u,v
      $\vec{s}^\star_v \leftarrow \alpha \vec{s}_v$                ▷ Multiply all bins by $\alpha$
      for $f \in \mathcal{F}$ do
          if value($f$) in bin j then
              $\vec{s}^\star_{vj} \leftarrow \vec{s}^\star_{vj} + (1-\alpha)$      ▷ Add (1-$\alpha$) to bin $j$
          end if
      end for
      $\vec{s}_u \leftarrow \beta \vec{s}_u + (1-\beta)\vec{s}^\star_v$     ▷ Updated summary of node $u$.
  end for
```

2.2.3 Choosing histogram bins. Essential hyperparameters of this method are the choices of the boundaries of the histograms bins. We propose to use one bin per category for categorical features. If the cardinality of a certain feature is too high, we propose to form bins using groups of categories. For numerical features, one can plot the distribution in the training data and choose sensible bin edges, for example on every 10th percentile of the distribution. The framework is not constrained by one choice of bins, as long as they can be updated in a streaming way.

2.3 Reducing space complexity

The space complexity of the graph-sprints approach (algorithm 1) is

$$M = |\mathcal{V}| \sum_{f \in \mathcal{F}} L_f \tag{9}$$

where $|\mathcal{V}|$ stands for the number of nodes, L_f stands for the number of bins of the histogram for feature f, and $\mathcal{F}$ stands for the set of features chosen to collect in histograms. In case this memory is too high, we propose the following methods to reduce memory further.

Reducing histogram size using similarity hashing Following the similarity hashing approach proposed in Jin et al. [10], we extend the method to the streaming setting. All histograms as defined in the previous sections are normalized (in the sense that bin values sum to 1), and we can concatenate them into one vector $\vec{s}_{tot}$. We can now define a hash mapping by choosing k random hyperplanes in $\mathbb{R}^M$ defined by unit vectors $\vec{h}_j$, $j = 1, \ldots, k$.

The inner product between the histograms vector and the k unit vectors results in a vector of k values, each value θ can be calculated using the dot product of the unit vector $\vec{h}_j$ and the histogram vector $\vec{s}_{tot}$, as illustrated in Equation 10. We use the superscript t to denote the current time step.

$$\theta^t_j = \vec{h}_j \cdot \vec{s}^t_{tot} \tag{10}$$

One can binarize the representation of the hashed vector using by taking the *sign* of the above θ^t_j.

Therefore, the resulting space complexity per node is k, replacing the number of bins in the memory M by the number of hash vectors k.

Importantly, the hashed histograms can be updated without storing any of the original histograms. Combining equations 4 and equation 10 and denoting $\vec{\delta}(\vec{f})$ the concatenation of the $\vec{\delta}$ vectors for all collected features, we get

$$\theta^{t+1}_j = \theta^t_j \cdot \alpha + \vec{h}_j \cdot \vec{\delta}(\vec{f}) \cdot (1-\alpha) \tag{11}$$

Therefore, we can compute the next hash θ^{t+1}_j or sign(θ^{t+1}_j) directly from the previous θ^t_j and the new incoming features $\vec{\delta}(\vec{f})$.It is also important to note that this hashing scheme is preserved when averaging.

Reducing embedding size using feature importance One can reduce the needed memory by relying on feature importance techniques. One possibility is to train a classifier on the raw node and/or edge features and determine feature importances, after which only the top important features are used in the *graph-sprints* framework. Or similarly train on all bins and decide the bins to be used based on their importance in the classification task.

3 EXPERIMENTS

3.1 Experimental setup

We assess the quality of the graph based features generated by the *graph-sprints* framework on node classification task. We use three publicly available datasets from the social and education domains, and two proprietary datasets from money laundering domain . We detail their main characteristics in Table 1 and Table 4, respectively. All datasets are CTDGs and are labeled. Each dataset is split into train, validation, and test sets respecting time (i.e., all events in the train are older than the events in validation, and all events in validation are older than the events in the test set). We use Optuna [1] to optimize the hyperparameters of all models, training 100 models using the TPE sampler and with 40 warmup trials. Each model trains using early stopping with a patience of 10 epochs, where the early stopping metric computed on the validation set is area under ROC curve.

3.1.1 Baselines. As a first baseline, we reproduce a state-of-the-art GNN model for CTDGs, the temporal-graph network (TGN) [20], which leverages a combination of memory modules and graph-based operators to obtain node representations. As an important note, we mention that the pytorch geometric [5] implementation of TGN was used, for which the sampling of neighbors uses a different strategy than the original TGN implementation. Indeed, the original paper allowed to sample from interactions within the same batch as long as they were older, while the pytorch geometric implementation does not allow within-batch information to be used. As also noted in the pytorch geometric documentation, we believe the latter to be more realistic. As a consequence, our TGN results are not directly comparable with the originally published TGN performances. In any case, the graph-sprints embeddings were computed using the same batch size and therefore also do not have access to within-batch information, allowing a fair comparison between the algorithms.

Two variations of the TGN architecture were used. First, TGN-attn was implemented, which was the most powerful variation in the original paper but is expected to be slower due to the graph-attention operations. Second, TGN-ID was implemented, which is a variation of the TGN where no graph-embedding operators are used, and only the embedding resulting from the memory module is passed to the classification layers.

A third baseline we use is Jodie [15]. We use the TGN implementation of Jodie, where instead of using Graph attention embeddings on top of the memory embedding, a time projection embedding module is used and where the loss function is otherwise identical to the TGN setting. For a fair comparison with TGN we use the same memory updater module, namely, gated recurrent units (GRUs).

The TGN-ID and Jodie baselines do not require sampling of neighbors, and were therefore chosen as lower-latency baselines compared to TGN-attn.

3.1.2 Graph-sprints and classifier. For each arriving edge, we apply the graph-sprints feature update (algorithm 1) to both the source node and the destination node in parallel. All edge features are used for the computation of the graph-sprints features, and for each feature bin edges are chosen as the 10 quantiles computed on the training data. Since the graph-sprints framework only creates features, a classifier is implemented for the classification tasks. We chose to implement a neural network consisting of dense layers and skip-connections every two layers. Hyperparameter optimization proceeds in two steps. First, default parameters for the classifier are used to optimize the discount factors of the *graph-sprints* framework, α and β. For this step, 50 models are trained. Subsequently, hyperparameter optimization of the classifier follows same approach as TGN, training 100 models.

Table 1: Information about public data [15].

	Reddit	Wikipedia	MOOC
#Nodes	10,984	9,227	7,047
#Edges	672,447	157,474	411,749
Label type	posting ban	editing ban	student drop-out
Positive labels	0.05%	0.14%	0.98%
Used split	75%-15%-15%	75%-15%-15%	60%-20%-20%

In all experiments, we test the following three cases. Firstly, we train the classifier using only raw features (Raw). We then train the classifier using only the graph-sprint features (GS). Finally, we train the classifier using both raw and graph-sprint features (GS+Raw).

3.1.3 Node Classification. For the node classification task on the Wikipedia, Reddit and Mooc datasets, we concatenate the source and destination node embeddings and feed the concatenated vector to the classifier, as is usual for datasets where labels are on the edge level. For the money laundering dataset labels are on the node level, hence we use the node embedding to predict its respective label.

3.2 Public dataset experiments

3.2.1 Task performance. In Table 2 we report the average test AUC ± std resulting from retraining the best model after hyperparameter optimization using 10 random seeds.

3.2.2 Inference runtime. We compare the latency of our framework to baseline GNN architectures. For this purpose, we run 200 batches of 200 events on the external datasets, Wikipedia, Mooc, and Reddit using the node classification task. We compute the average time over 10 runs. Both models were running on Linux PC with 24 Intel Xeon CPU cores (3.70GHz) and a NVIDIA GeForce RTX 2080 Ti GPU (11GB).

As depicted in Figure 2, our graph-sprints consistently outperforms other baselines (TGN-attn, TGN-ID, Jodie) in the node classification task while also demonstrating a significantly lower inference latency. Compared to TGN-attn, the GS achieves better classification results but is up to approximately an order of magnitude faster (Figure 2).

Furthermore, the speedups achieved by graph-sprints are expected to be significantly higher in a big-data context, where the data is stored in a distributed manner rather than in memory as in our current experiments. In such scenarios, graph operations used in graph-neural networks like TGN-attn would incur even higher computational costs.

Table 2: Results on Node classification task, we report the average test AUC ± std achieved by retraining the best model after hyperparameter optimization using 10 random seeds. Our models, Raw, GS, and GS+Raw, use the same ML classifier but differ in the features employed for training. Raw uses raw edge features, GS uses graph-sprints histograms, and GS+Raw combines both. We identify the best model and highlight the second best model.

Method	AUC ± std		
	Wikipedia	Mooc	Reddit
Raw	58.5 ± 2.2	62.8 ± 0.9	55.3 ± 0.8
TGN-ID	88.9 ± 0.2	63.0 ± 17	61.3 ± 2.0
Jodie	87.2 ± 0.9	63.7 ± 16.7	61.9 ± 2.0
TGN-attn	86.6 ± 2.8	75.8 ± 0.4	69.0 ± 0.9
GS	**91.2 ± 0.4**	74.7 ± 0.1	**71.3 ± 2.1**
GS+Raw	89.6 ± 0.4	**76.3 ± 0.1**	66.1 ± 0.6

Recently, APAN [28] has attempted to build a low-latency framework for CTDGs. Their approach consisted of performing the expensive graph operations asynchronously, out of the inference loop. In that way, they achieved inference speeds of 4.3ms per batch on the Wikipedia dataset, but we cannot directly compare those results with ours (GS: 1.4ms, TGN-attn: 6ms) due to the different setup and hardware. Importantly, their approach achieves low-latency by sacrificing up-to-date information at inference time. Indeed, the inference step is performed without access to the most recent embeddings, because the expensive graph operations to compute the embeddings are performed asynchronously.

3.2.3 Memory reduction. Both the Wikipedia and Reddit datasets consist of 172 edge features. By calculating graph-sprints with 10 quantiles per feature, along with incorporating in/out degrees histograms and time-difference histograms, we obtain a node embedding of 1742 features (one feature per histogram bin). In our experimental setup, we concatenate the source and destination node embeddings for source label prediction, similar to SotA approaches, resulting in a 3484-feature vector. To reduce the size of the node embeddings, we propose a similarity hashing-based memory reduction technique (Section 2.3)). Our experiments, as presented in Table 3, demonstrate that this technique can significantly reduce storage requirements without affecting the AUC in the node classification task. For the Wikipedia dataset, storage usage can be reduced to just 0.12% without any impact on AUC. In the case of the Reddit dataset, storage can be reduced to 25% with a 1% AUC sacrifice or to 10% with a 2% AUC sacrifice. The reduction percentage can be fine-tuned as a hyperparameter, considering the use case and dataset.

Table 3: Effect of memory reduction on the node classification task. Average test AUC ± std resulting from retraining the best model after optimization using 10 random seeds. We use only the GS embeddings to train the classifier. Since MOOC has less features we add '-' where number of features < 1.

Space used	Wikipedia	Mooc	Reddit
100%	91.2 ± 0.4	72.7 ± 0.1	71.3 ± 2.1
50%	91.1 ± 0.1	73.9 ± 1.0	71.1 ± 1.5
25%	91.2 ± 0.4	72.8 ± 0.9	70.3 ± 1.8
10%	91.3 ± 0.1	72.2 ± 0.3	69.0 ± 1.5
0.5%	90.7 ± 0.3	-	65.1 ± 2.1
0.12%	91.0 ± 0.1	-	54.6 ± 3.9

3.3 Anti-money laundering (AML) experiments

In money laundering, the criminals' objective is to hide the illegal source of their money by moving funds between various accounts and financial institutions. In these experiments, our objective is to enrich an classifier with graph-based features generated by our *graph-sprints* framework and evaluate whether that enhances the model detection of suspicious activity.

3.3.1 Datasets. We evaluate the graph-sprints framework in the AML domain using two real-world banking datasets. Due to privacy concerns, we can not disclose the identity of the financial institutions (FIs) nor provide exact details regarding the node features. We refer to the datasets as *FI-A* and *FI-B*. The graphs in this use-case are constructed by considering the accounts as nodes and the money transfers between accounts as edges. Table 4 shows approximate details of these datasets.

Table 4: Information about AML datasets.

	FI-A	FI-B
#Nodes	≈400000	≈10000
#Edges	≈500000	≈2000000
Positive labels	2-5%	20-40%
Duration	≈300 days	≈600 days
Edges/day (mean ± std)	1500 ± 750	3000 ± 5000
Used split	60%-10%-30%	60%-10%-30%

3.3.2 Results. For privacy reasons, we can not disclose the entire models' ROC curve; instead, we report relative improvements in ROC curves when compared against a baseline model that does not use graph features. In this way, the baseline models correspond to the $\Delta AUC = 0$, while any gain in Recall compared to the baselines results in positive values of the ΔAUC.

In Table 5, we show the ΔAUC, we obtain an improvement of around 4.4%AUC in the *FI-A dataset*, and an improvement of around 25.5% AUC in *FI-B dataset*.

Table 5: Node classification results in AML data. We report relative gain in AUC (ΔAUC) compared to a baseline model that does not utilize graph features. We report the average test ΔAUC ± std achieved by retraining the best model after hyperparameter optimization using 10 random seeds. We identify the best model and highlight the second best model.

Method	ΔAUC ± std	
	FI-A	FI-B
TGN-ID	+0.1 ± 0.1	+24.4 ± 0.2
Jodie	+0.0 ± 0.1	+24.5 ± 0.2
TGN-attn	+0.3 ± 0.7	+25.1 ± 0.3
GS	+2.0 ± 0.2	**+25.5 ± 1.4**
GS+Raw	**+4.4 ± 0.3**	+19. ± 1.4

4 RELATED WORK

Our graph-sprints approach handles real-time information and has low latency; other approaches typically require trade-offs between these two dimensions (Figure 1). We discuss various related methods below, depending on whether they sample graphs using random-walks or k-hop neighborhoods.

4.1 Random-walk based methods

DeepWalk [19] and node2vec [7] are two random-walk based methods to extract node embeddings on static graphs. DeepWalk leverages random-walks to generate node representations using the skip-Gram method [17], by treating walks as the equivalent of sentences. Node2vec extends DeepWalk to perform biased random-walks. The main limiting factors of these methods are that they disregard node and edge features as well as temporal information, since they are designed for static graphs. Sajjad et al. [21]. extend these random-walk based methods to discrete-time dynamic graphs. For every new time step, the authors propose to identify which random-walks from the previous graph snapshot are affected by changes in the current graph snapshot. Then, these random-walks are either fully or partially re-generated. While some efficiency is gained with the proposed method, it is far from applicable to CTDGs and in low-latency settings.

Node2bits [10] leverages temporal random-walks as a backbone to compute node embeddings in CTDGs, similarly to our random-walk based approach. It considers the temporal information by defining several time windows over the sampled random-walks, and aggregates node attributes in these time windows into histograms. Finally, the authors propose to compresses and binarizes the node embeddings using similarity hashing. Node2bits is tailored for the user stitching problem, does not include edge-features and performs costly computations which cannot be performed with low latency.

Continuous-time Dynamic Node Embeddings (CTDNE) [16, 18] were proposed to generate time-aware embeddings, generalizing the node2vec framework to CTDGs. The authors consider the graph as a stream of edges, and propose to perform temporal walks starting from seed nodes chosen from a temporally biased distribution. The authors propose to include the extracted temporal walks in a downstream framework such as skip-Gram or a GNN, in order to learn time-aware node embeddings. Similarly, temporal random-walks have been used to extract embeddings into hyperbolic spaces [27]. Causal anonymous walks [29] propose to use anonymized walks in order to encode motif information. Similarly, NeurTWs [13] explicitly model time in the anonymous walks using Neural Ordinary Differantial Equations (NeuralODEs). Unlike our graph-sprints framework, full random-walks need to be performed. For a deeper analysis of the trade-offs between accuracy and run-time complexity in random-walk based approaches, we refer the reader to the study by Talati et al. [24].

4.2 K-hop neighborhood based methods

Most GNN-based methods require a K-hop neighborhood on which message-passing operations lead to node embeddings [2, 9, 14, 25]. To deal with CTDGs, a simple approach is to consider a series of discrete snapshots of the graph over time, on which static methods are applied. Such approaches however do not take time properly into account and several works propose techniques to alleviate this issue[6, 12, 22] .

To better deal with CTDGs, other works focus on including time-aware features or inductive biases into the architecture. Deep-Coevolve [3] and Jodie [15] train two RNNs for bipartite graphs, one for each node type. Importantly, the previous hidden state of one RNN is also added as an input to the other RNN. In this way, the two RNNs interact, in essence performing single-hop graph aggregations. TGAT [32] proposes to include temporal information in the form of time encodings, while TGN [20] extends this framework and also includes a memory module taking the form of a recurrent neural network. In [11], the authors replace the discrete-time recurrent network of TGN with a NeuralODE modeling the continuous dynamics of node embeddings.

APAN [28] proposes to reduce the latency at inference time by decoupling the more costly graph operations from the inference module. The authors propose a more light-weight inference module that computes the predictions based on a node's embedding as well as a node's mailbox, where the mailbox contains messages constructed from the embeddings of recently interacting nodes. The mailbox is updated asynchronously, i.e. separated from the inference module, and involves the more expensive k-hop message passing. While APAN improves the latency at inference time, it sacrifices some memory since each node's state is now expanded with a mailbox and it potentially uses outdated information at inference time due to asynchronous update of this mailbox. Also towards reducing computational costs of GNNs, HashGNN [30] leverages MinHash to generate node embeddings suitable for the link prediction task, where nodes that results in the same hashed embedding are considered similar. SGSketch [33] is a streaming node embedding framework that similarly to our strategy uses gradual forgetting mechanism to gradually forget outdated edges, achieving significant speedups. Differently than our approach SGS-ketch uses the gradual forgetting strategy to update the adjacency matrix and therefore only considers the graph structure and does not consider node attributes.

5 CONCLUSIONS

We described graph-sprints, a framework to compute time-aware embeddings for CTDGs with low latency. We showed how the graph-sprints features, combined with a neural network classifier, obtain better predictive performance compared to state-of-the-art methods, while being at least 4 times faster at inference. Interestingly, our method is not based on graph neural networks, but instead proposes to characterize feature distributions of neighborhoods in streaming histograms. Topological information is included by summarizing degree distributions. In future work, it would be interesting to extend the graph-sprints framework to heterogeneous graphs, and explore how GNNs could inherit some of the strengths of graph-sprints.

REFERENCES

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2623–2631.

[2] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. 2020. Principal neighbourhood aggregation for graph nets. *Advances in Neural Information Processing Systems* 33 (2020), 13260–13271.

[3] Hanjun Dai, Yichen Wang, Rakshit Trivedi, and Le Song. 2016. Deep coevolutionary network: Embedding user and item features for recommendation. *arXiv preprint arXiv:1609.03675* (2016).

[4] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The world wide web conference*. 417–426.

[5] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[6] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. 2018. Dyngem: Deep embedding method for dynamic graphs. *arXiv preprint arXiv:1805.11273* (2018).

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

[8] Jin Guo, Zhen Han, Zhou Su, Jiliang Li, Volker Tresp, and Yuyi Wang. 2022. Continuous Temporal Graph Networks for Event-Based Graph Data. *arXiv preprint arXiv:2205.15924* (2022).

[9] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[10] Di Jin, Mark Heimann, Ryan A Rossi, and Danai Koutra. 2019. Node2bits: Compact time-and attribute-aware node representations for user stitching. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 483–506.

[11] Di Jin, Sungchul Kim, Ryan A Rossi, and Danai Koutra. 2020. From static to dynamic node embeddings. *arXiv preprint arXiv:2009.10017* (2020).

[12] Di Jin, Sungchul Kim, Ryan A Rossi, and Danai Koutra. 2022. On Generalizing Static Node Embedding to Dynamic Settings. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. 410–420.

[13] Ming Jin, Yuan-Fang Li, and Shirui Pan. 2022. Neural Temporal Walks: Motif-Aware Representation Learning on Continuous-Time Dynamic Graphs. In *Advances in Neural Information Processing Systems*.

[14] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[15] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *Proceedings of the 25th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM.

[16] John Boaz Lee, Giang Nguyen, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. 2020. Dynamic node embeddings from edge streams. *IEEE Transactions on Emerging Topics in Computational Intelligence* 5, 6 (2020), 931–946.

[17] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. https://doi.org/10.48550/ARXIV.1301.3781

[18] Giang Hoang Nguyen, John Boaz Lee, Ryan A Rossi, Nesreen K Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-time dynamic network embeddings. In *Companion proceedings of the the web conference 2018*. 969–976.

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

[20] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In *ICML 2020 Workshop on Graph Representation Learning*.

[21] Hooman Peiro Sajjad, Andrew Docherty, and Yuriy Tyshetskiy. 2019. Efficient representation learning using random walks for dynamic graphs. *arXiv preprint arXiv:1901.01346* (2019).

[22] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th international conference on web search and data mining*. 519–527.

[23] Amauri Souza, Diego Mesquita, Samuel Kaski, and Vikas Garg. 2022. Provably expressive temporal graph networks. *Advances in Neural Information Processing Systems* 35 (2022), 32257–32269.

[24] Nishil Talati, Di Jin, Haojie Ye, Ajay Brahmakshatriya, Ganesh Dasika, Saman Amarasinghe, Trevor Mudge, Danai Koutra, and Ronald Dreslinski. 2021. A deep dive into understanding the random walk-based temporal graph learning. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 87–100.

[25] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[26] Jianian Wang, Sheng Zhang, Yanghua Xiao, and Rui Song. 2021. A review on graph neural network methods in financial applications. *arXiv preprint arXiv:2111.15367* (2021).

[27] Lili Wang, Chenghan Huang, Weicheng Ma, Ruibo Liu, and Soroush Vosoughi. 2021. Hyperbolic node embedding for temporal networks. *Data Mining and Knowledge Discovery* 35, 5 (2021), 1906–1940.

[28] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 2021. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*. 2628–2638.

[29] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive representation learning in temporal networks via causal anonymous walks. *arXiv preprint arXiv:2101.05974* (2021).

[30] Wei Wu, Bin Li, Chuan Luo, and Wolfgang Nejdl. 2021. Hashing-accelerated graph neural networks for link prediction. In *Proceedings of the Web Conference 2021*. 2910–2920.

[31] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[32] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962* (2020).

[33] Dingqi Yang, Bingqing Qu, Jie Yang, Liang Wang, and Philippe Cudre-Mauroux. 2022. Streaming graph embeddings via incremental neighborhood sketching. *IEEE Transactions on Knowledge and Data Engineering* 35, 5 (2022), 5296–5310.

[34] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. 2021. Graph neural networks and their current applications in bioinformatics. *Frontiers in genetics* 12 (2021), 690049.