Tunable Streaming Graph Embeddings at Scale

Serafeim Papadias
Technische Universität Berlin
supervised by Prof. Volker Markl
s.papadias@tu-berlin.de

ABSTRACT

An increasing number of real-world applications require machine learning tasks over large-scale streaming graphs, where nodes and edges are continuously being added or deleted. Graph embeddings have been widely used for solving such tasks by capturing the graph structure and features into a low-dimensional latent space. However, current approaches have one or more of the following disadvantages: (i) they are designed for either static or dynamic graphs and thus, need retraining after each graph change or periodically updating the embeddings after each snapshot arrival, (ii) they fail to scale to today's size of graphs composed of billions of nodes, or (iii) yet the ones devised for streaming graphs perform redundant retraining computations by mandating continuous embedding updates even if the accuracy is not improved. The goal of this thesis is to overcome the abovementioned problems by devising *tunable* streaming methods that can scale to massive graphs. We envision an end-to-end ML streaming system that achieves that goal and provides users with abstractions to easily define their own streaming embedding algorithms.

1. INTRODUCTION

Graphs are omnipresent in various domains such as social media, transportation, finance, IoT, and biological networks. Typically, real-world graphs are inherently dynamic, entailing continuous additions and deletions of vertices and edges. The frequency of these graph updates lies on a spectrum. In *dynamic* graphs, updates appear in batches as graph snapshots and are applied periodically. In *streaming* graphs, updates arrive spontaneously and are incorporated on-the-fly. For instance, social networks that model friendships between users are highly dynamic, while citations networks modelling relations among scholars in academic networks are less dynamic. Many real-world applications, such as news recommendation or crime detection, can be modeled as machine learning (ML) tasks over streaming graphs. Characteristic tasks include vertex classification, link prediction, link reconstruction, topic modeling, and, community, anomaly, fraud, and outlier detection.

A very popular and effective technique for solving such ML tasks are graph node embeddings (a.k.a network representation learning). Given a graph $G = (V, E)$ with n nodes, a graph embedding maps each node $v \in G$ to a compact feature vector in a lower k dimentional space ($k \ll n$), which captures graph structure and properties in the vicinity of v. These vectors are derived by optimizing objective functions preserving geometric relationships among graph nodes. Computing embeddings is a crucial problem by itself, as they serve as inputs to downstream ML tasks mentioned above.

The frequency of updating the embedding vectors plays a significant role on runtime performance and accuracy, and should be driven by the subsequent ML task. For example, critical downstream applications, such as anomaly detection, should instantly react to graph changes for capturing all anomalies; hence, their input embeddings must constantly remain up-to-date for producing accurate results. For the above scenario, static embedding methods, such as [9], are unsuitable, as they need to retrain embeddings from scratch after each graph change. Dynamic techniques, such as [6] are not sufficient either, as they update embeddings periodically after each snapshot arrival; hence, fall short of discovering anomalies appearing during the idle period between two graph snapshots. Even though streaming algorithms [7, 10] refine the embedding vectors after every graph update, this can be potentially unecessary as graph structure may not be substantially altered to affect the accuracy of the downstream ML task. Thus, anomaly detection and similar continuous ML tasks dictate flexible streaming solutions that are *tunable*: they can adjust the frequency with which the embeddings are being updated.

Nowadays, real-world graphs not only dictate (tunable) streaming algorithms due to their ever-changing nature, but also *scalable* solutions because of their massive size. However, the majority of existing embedding techniques, either the ones concerning static graphs [3, 9, 11, 12] or the ones designed for dynamic graphs [6, 7, 10], are centralized; hence, they do not scale on massive graphs. Ideally, scalability should be achieved through the distribution of processing on clusters of commodity machines; thus, avoiding solutions that utilize expensive servers machines with several terabytes of main memory or unaffordable GPUs. Few available scalable graph embeddings systems [5] exist; however, they are unable to operate on evolving graphs. Thus, there is a lack of solutions that can both achieve scalability and

Proceedings of the VLDB 2020 PhD Workshop, August 31st, 2020. Tokyo, Japan.

conduct streaming graph embedding processing.

As ML and graph embeddings in particular become more and more popular there is a need for systems facilitating the development of such algorithms by hiding the system complexity from the users. For instance, how the system distributes the processing of defined algorithms should be agnostic to the users. There are few systems with this goal: they provide primitives for distributed random walk computation on static graphs [13] or for graph neural network computation [14, 15]. However, no such comprehensive system for (tunable) streaming graph embeddings exists.

In this thesis, we strive to: (i) devise *tunable* streaming methods that generate embeddings incrementally by adjusting the frequency of vector updates, (ii) build a scalable solution which processes streaming embedding algorithms in a distributed manner, and (iii) provide abstractions that ideally incorporate distinct classes of streaming embedding methods. Our goal is to synthesize these solutions into KAIXIS[1], a novel end-to-end system, capable of bridging the gap between streaming and distributed embedding methods.

2. RELATED WORK

Representation learning on graphs received huge attention from researchers in the past few years. There are three main categories of graph embeddings based on: (a) random-walks, (b) matrix-factorization, and (c) deep learning. In what follows, we focus on the first category as the other two incur extremely high costs rendering them unsuitable for streaming scenarios, i.e., deep learning-based train directly on the whole graph and factorization-based suffer from expensive matrix operations [6]. We mainly review random walk-based (dynamic) network embedding algorithms and systems.

Walk-based Algorithms. Embedding methods based on random walks mainly consist of two phases: the *random walking* that explores, samples and captures certain properties of the graph, and the *training* that subsequently ingests the produced random walks and trains embedding vectors e.g., using the well-known Skip-Gram [8] model. Depending on the random walk type, different properties are captured. Specifically, DeepWalk [9] performs truncated random walks to preserve first-order proximities in a graph, whereas node2vec [3] deploys biased second-order walks capturing second-order proximities. LINE [11] optimizes an objective function that preserves both first-order and second-order proximities while deriving the embedding vectors. GraRep [1] captures higher-order proximities in the final embedding vectors. GraphCSC [2] deploys centrality-based walks capable of learning embeddings that preserve graph characteristics, such as degree and betweenness, and finally aggregates them into one vector. Nevertheless, all the techniques above are designed for static graphs; hence, are unable to adapt in streaming scenarios that we focus on.

Interestingly enough, only [7, 10] address the problem of streaming graph embeddings. In [7], a rather ad-hoc influence propagation model is used for locating nodes whose embedding vector is influenced by a graph change (either addition or deletion). Also, a *vertex stream* is assumed, i.e., each new node arrives along with its complete adjacency list, which is a limitation. In [10], a method that incrementally maintains first and second-order random walks in a streaming graph is proposed; however, it is centralized and lacks of theoretical guarantees of correctness.

[1]From the turkish word Kayikçi; the owner of a fishing boat.

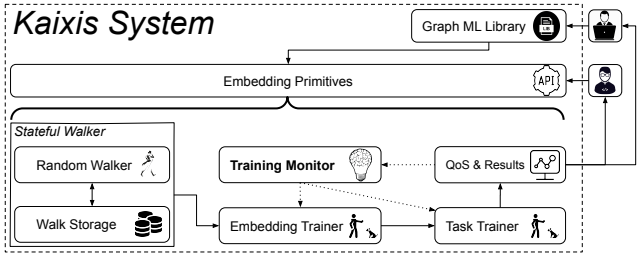

Figure 1: System architecture of Kaixis.

Walk-based embedding algorithms fall short mainly in two crucial aspects. First, they incur computational costs proportional to the number of deployed walks; rendering them insufficient for massive graphs, especially the widespread centralized solutions. Instead, massively-parallel and ideally cost-effective solutions, such as distributing tasks into clusters of commodity machines, are required. Secondly, there is no well-established streaming walk-based technique, which has robust theoretical analysis. In contrast, we aim for (tunable) streaming and distributed walk-based embeddings that are also theoretically established.

Walk-based Systems. KnightKing [13] is a distributed system specific for computing random walks on static graphs. It offers a walker-centric computation model, which is able to express various walk algorithms. Its extreme efficiency is attributed to the rejection sampling it utilizes, especially when computing cumbersome higher-order walks. However, KnightKing is incapable of computing streaming random walks, as it requires the whole graph in advance. Except for academia, industry has also shown huge interest on scalable systems for the graph neural networks (GNNs) [14, 15], which are relevant to graph embeddings. Aligraph [15] provides primitive operators that abstract common concepts among GNN algorithms, facilitating users in implementing and deploying them. AGL [14] builts upon k-hop neighborhoods[2] and utilizes MapReduce and Parameters Servers to speed up GNN training; a relatively straightforward task due to the linear nature of GNNs. As far as we know, there is no comprehensive system that provides *primitives* for streaming embedding algorithms.

3. KAIXIS ARCHITECTURE

In this section we introduce KAIXIS, our envisioned end-to-end system for computing streaming random walk-based graph embeddings on the fly and at scale. Figure 1 shows the system architecture consisting of seven main components: a Graph ML Library, the Embedding Primitives, a Stateful Walker, an Embedding Trainer, a Task Trainer, a Training Monitor, and the Results and QoS module.

The input of the system consists of continuous ML tasks, such as anomaly detection, along with a graph stream source and certain user-defined requirements (e.g., accuracy $> 70\%$). KAIXIS perpetually computes tunable streaming graph embedding vectors, which are subsequently forwarded as inputs

[2]For vertex v, the set of vertices reachable from v within at most k steps.

to high stakes ML tasks. The output of the system consists of (i) real-time results for the high-stakes tasks, and (ii) quality of service (QoS) metrics, as depicted in Figure 1. KAIXIS addresses two types of users: *end-users* and *developers*. End-users interact with the system through the Graph ML Library for specifying their ML task and QoS requirements. Developers use the flexible Embedding Primitives for defining tunable streaming embedding algorithms. In a nutshell, after users submit their queries, KAIXIS follows concrete steps. Namely, the Stateful Walker constantly explores the evolving graph and keeps the stored random walks up-to-date. Subsequently, the Embedding Trainer updates the existing embedding vectors on-the-fly, based on the walks received from the Stateful Walker. Then, the Task Trainer uses the embeddings to produce query results. The Training Monitor has the power to switch on and off either of the Embedding and Task Trainer. It *selectively* enables retraining of existing embeddings and/or ML task models only when needed to avoid excessive computation. In the following, we detail each component.

Graph ML Library. End-users interact with KAIXIS via this library, which is a collection of possibly pre-configured algorithmic operators directly invoked without necessitating hyper-parameter tuning. These operators solve tasks, such as link prediction, anomaly detection, fraud detection, outlier detection, graph reconstruction and vertex classification.

Embedding Primitives. Developers interact with KAIXIS through the Embedding Primitives, which enable them to easily implement, integrate and deploy streaming embedding methods, without having to worry about how the distribution is handled by KAIXIS. Additionally, any optimized implementation of a primitive, also speeds up every method that utilizes this primitive. Hence, Embedding Primitives unify optimizations of distinct embedding methods.

Stateful Walker. Random walks are core concepts of graph embeddings. A Stateful Walker in KAIXIS, consists of two parts: the Random Walker and the Walk Storage. The former constantly explores the ingested graph stream and produces new random walks for newly appearing nodes or updates walks attributed to already existing nodes. The Walk Storage unit is responsible for efficiently storing the latest *random walk corpus*.

Embedding Trainer. As shown in Figure 1, the random walks produced (either newly formed or new parts of modified ones) are forwarded to the Embedding Trainer to refine and output the latest embedding vectors by conducting incremental training. In doing so, the trainer hosts a variety of training algorithms in its artillery e.g., online Skip-Gram and Stochastic Gradient Descent models.

Task Trainer. The user's selected ML application e.g., anomaly detection, has to be executed in a continuous way. We thus opt for online ML algorithms where the training of the model is performed in an online fashion similarly with the serving part. KAIXIS deploys the Task Trainer to update on-the-fly the model of the specified ML task and finally output the *prediction* results.

Training Monitor. Both Embedding and Task Trainer perform online training. However, it is important to note that performing *blindly* online training, i.e., updating the model after every single change in the streaming graph may lead to unnecessary excessive computation and thus, degrade the system's performance. Specifically, the frequency of Task Trainer should be large enough to satisfy the user-defined requirements. The frequency of Embedding Trainer should be driven by the downstream (possibly critical) ML tasks, such that the embeddings are kept up-to-date. Thus, the embedding training is not everlasting but tuned in real-time by the Training Monitor. For instance, if the user wants to detect outliers in an evolving graph, the embedding vectors should be updated just as frequently as it is necessary for capturing all outliers instantly. In other words, if an update in the embedding vector does not yield any change in the prediction results, it should not be performed to avoid unnecessary computation.

Results & QoS. This component serves as a reporting unit gathering results and QoS metrics of the ML task from the Task Trainer. QoS consist of *accuracy* metrics, e.g., area under the curve (AUC) and micro-F1 score, and *performance* measurements, such as throughput and execution time.

4. THE RESEARCH ROAD AHEAD

Our goal is to serve dynamic applications that can leverage graph embeddings used for retrieving information from an evolving graph. In essence, KAIXIS extracts embeddings from a graph stream in a tunable way and feeds them to downstream ML tasks. The system can operate at the finest granularity, i.e., always derive the latest vectors and train the latest ML model of a task. However, its profound goal is actually to avoid excessive training and instead strive for continuously adjusting retraining frequency by monitoring the QoS metrics.

4.1 Research Challenges

Realizing KAIXIS is far from straightforward. Below, we highlight five research challenges:

(1) Streaming Random Walks. Maintenance of random walks on evolving graphs should not compute all walks from scratch after a graph update, but only revise the already kept walk corpus. Most importantly, the refined walk corpus at time $t+1$ should be *statistically equivalent* to the corpus at time t. Equivalently, the updated walk corpus at time $t+1$ should have the same probability of being produced as a corpus derived from scratch by totally recalculating all the walks. Different policies can be used for updating walks [10], but clearly much remains to be done; both on the theoretical side for coming up with sound policies and on the performance side via distribution.

(2) Scalable Random Walks. The huge magnitute and the high dynamicity of nowadays graph data renders centralized random walk calculation highly insufficient. Yang et al. [13] crafted a whole system solely dedicated to *distributed* random walks computation. Adapting their ideas to streaming graphs is far from trivial. One cannot afford to store the entire graph stream in a streaming setting. In addition, one should carefully distribute the graph on-the-fly to facilitate the walk calculation by cluster nodes, while avoiding excessive communication. Network communication is too costly, therefore acute *streaming graph partitioning* should ensure extremely scalable streaming random walks.

(3) Walk Storage Sharing. In graph node embeddings, numerous random walks are created for each single vertex, resulting in walk sets with *overlapping* parts. Since KAIXIS needs to maintain a *random walk corpus*, effective techniques that store overlapping walk parts only once are crucial. The

real challenge, is to come up with compression schemes for succinct representation of the whole walk corpus. Ideally, the compression should be lossless and enable processing of walks in their compressed form without the need for de-serialization. Finally, the streaming setting increases the complexity of the problem, as it dictates the possibility of updating the walks while in compressed form.

(4) Monitoring. The Training Monitor is the brain of our conceived system: It tunes the frequency of retraining performed by either the Embedding Trainer or the Task Trainer. A number of research questions arise, such as: (i) *when* should the Training Monitor trigger the Embedding and Task Trainers, e.g., periodically or based on a certain reasonable mechanism, (ii) *what* is the impact of a trainer that is *disabled* to the final ML task result, and (iii) *how* each specific ML task chosen affects the monitor's decisions, i.e., high stakes ML tasks would differ from non-critical ones. Clearly, the monitor's behaviour is driven by downstream ML tasks.

(5) Primitives. To facilitate users implementing, integrating and deploying streaming embedding methods, Kaixis should offer primitives that hide the implementation details of how distribution is handled. Designing such primitives is challenging as it implies breaking various embedding algorithms down to "atoms". In Kaixis, primitives are executed as extremely performant streaming and distributed random walk-based operations. These abstractions offer the potential for transparent optimizations, i.e., optimizations to a primitive used by many algorithms, end up optimizing them all in one shot.

4.2 Research Plan

To conclude, we present our research strategy for tackling the aforementioned challenges and realizing Kaixis.

Streaming and Scalable Random Walks. The first step is to design the Stateful Walker for calculating streaming random walks using streaming graph partitioning and distribution. To assist our goal we plan to use an efficient and succinct walk storage representation. Finally, we plan to establish the Stateful Walker theoretically by: (i) proving the statistical equivalence of walk update policies, and (ii) deriving complexity bounds for procedures updating walks.

Monitored Embedding Training. On the one hand, we plan to investigate relevant online training algorithms proposed in the literature (e.g. [4]) and thoroughly evaluate them to decide which ones to incorporate in the Embedding Trainer's artillery. On the other hand, we plan to design the Training Monitor such that it configures frequency of retraining/updating embeddings, driven by the importance of the downstream ML tasks; for high stakes tasks, this frequency is larger. To achieve scalability, both the Embedding Trainer and the Training Monitor should be carefully conjured to operate in a fully distributed manner, also aiming on minimizing communication between them.

Powerful Primitives. Our perpetual goal along this endeavour is to conjure expressive primitive abstractions that facilitate users. Finally, we strive for devising effective optimizations for each primitive, as various methods enclosing such a primitive will benefit simultaneously.

Acknowledgments. The author would like to thank Prof. Volker Markl and Dr. Zoi Kaoudi for their pristine guidance, as well as Dr. Eleni Tzirita Zacharatou for her invaluable feedback. This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).

5. REFERENCES

[1] S. Cao, W. Lu, and Q. Xu. GraRep: Learning Graph Representations with Global Structural Information. In *CIKM*, pages 891–900, 2015.

[2] H. Chen, H. Yin, T. Chen, Q. V. H. Nguyen, W. Peng, and X. Li. Exploiting Centrality Information with Graph Convolutions for Network Representation Learning. In *ICDE*, pages 590–601, 2019.

[3] A. Grover and J. Leskovec. Node2vec: Scalable Feature Learning for Networks. In *KDD*, page 855–864, 2016.

[4] N. Kaji and H. Kobayashi. Incremental Skip-gram Model with Negative Sampling. In *EMNLP*, pages 363–371, 2017.

[5] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *SysML*, page 285–296, 2019.

[6] J. Li, H. Dani, X. Hu, J. Tang, Y. Chang, and H. Liu. Attributed Network Embedding for Learning in a Dynamic Environment. In *CIKM*, page 387–396, 2017.

[7] X. Liu, P.-C. Hsieh, N. Duffield, R. Chen, M. Xie, and X. Wen. Real-Time Streaming Graph Embedding Through Local Actions. In *WWW*, page 285–293, 2019.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*, pages 3111–3119, 2013.

[9] B. Perozzi, R. Al-Rfou, and S. Skiena. DeepWalk: Online Learning of Social Representations. In *KDD*, page 701–710, 2014.

[10] H. P. Sajjad, A. Docherty, and Y. Tyshetskiy. Efficient Representation Learning Using Random Walks for Dynamic Graphs. *CoRR*, abs/1901.01346, 2019.

[11] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. LINE: Large-Scale Information Network Embedding. In *WWW*, page 1067–1077, 2015.

[12] D. Wang, P. Cui, and W. Zhu. Structural Deep Network Embedding. In *KDD*, page 1225–1234, 2016.

[13] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang. KnightKing: A Fast Distributed Graph Random Walk Engine. In *SOSP*, page 524–537, 2019.

[14] D. Zhang, X. Huang, Z. Liu, Z. Hu, X. Song, Z. Ge, Z. Zhang, L. Wang, J. Zhou, and Y. Qi. AGL: a Scalable System for Industrial-purpose Graph Machine Learning. *arXiv preprint arXiv:2003.02454*, 2020.

[15] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. AliGraph: A Comprehensive Graph Neural Network Platform. *VLDB*, 12(12):2094–2105, 2019.