EVOLVING NETWORK REPRESENTATION LEARNING BASED ON RANDOM WALKS

FARZANEH HEIDARI

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF COMPUTER SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
JULY 2019

Abstract

Large-scale network mining and analysis is key to revealing the underlying dynamics of networks, not easily observable before. Lately, there is a fast-growing interest in learning low-dimensional continuous representations of networks that can be utilized to perform highly accurate and scalable graph mining tasks. A family of these methods is based on performing random walks on a network to learn its structural features before feeding the sequence of random walks in a deep learning architecture to learn a network embedding. While these methods perform well, they can only operate on static networks. However, in real-world, networks are evolving, as nodes and edges are continuously added or deleted. As a result, any previously obtained network representation will now be outdated having an adverse effect on the accuracy of the network mining task at stake. The naive approach to address this problem is to re-apply the embedding method of choice every time there is an update to the network. But this approach has serious drawbacks. First, it is inefficient, because the embedding method itself is computationally expensive. Then, the network mining task outcome obtained by the subsequent network representations are not directly comparable to each other, due to the randomness involved in the new set of random walks involved each time. In this research, we propose a random-walk based method for learning representations of evolving networks (EVONRL). The key idea of our approach is to maintain a set of random walks that are consistently valid with respect to the updated network topology. That way we are able to continuously learn a new mapping from the evolving network to a low-dimension network representation, by only updating a small number of random walks required to re-obtain an embedding. Moreover, we present an analytical method for determining the right time to obtain a new representation of the evolving network balancing accuracy and time performance. A thorough experimental evaluation is performed that demonstrates the effectiveness of our method against sensible baselines, for a varying range of conditions.

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Manos Papagelis for the continuous support of my study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. Also, I wish to thank the members of my dissertation committee: Dr. Hui Jiang and Dr. Marin Litoiu for generously offering their time and guidance throughout the review of this document.

Table of Contents

List of Tables

List of Figures

$ifdefined@prefacefile$

Preface

ifdefined@abbreviationsfile

Abbreviations

1 Introduction

Network science, built on the mathematics of graph theory, leverage network structures to model and analyze pairwise relationships between objects (or people) [31]. With a growing number of networks – social, technological, biological – becoming available and representing an ever increasing amount of information, the ability to easily and effectively perform *large-scale network mining and analysis* is key to revealing the underlying dynamics of these networks, not easily observable before. Traditional approaches to network mining and analysis inherit a number of limitations. First, networks are typically represented as adjacency matrices, which suffer from high-dimensionality and data sparsity issues. Then, network analysis typically requires domain-knowledge in order to carry out the various steps of network data modeling and processing that is involved, before (multiple iterations of) analysis can take place. An ineffective network representation along with a requirement for domain expertise, render the whole process of network mining cumbersome for non-experts and limits their applicability to smaller networks.

1.1 State-of-the-art and Limitations

To address the aforementioned limitations, there is a growing interest in learning *low-dimensional representations of networks*, also known as *network embeddings*. These representations are learned in an agnostic way (without domain-expertise) and have the potential to improve the performance of many downstream network mining tasks that now only need to operate in lower dimensions. Example tasks include *node classification*, *link prediction* and *graph reconstruction* [46], to name a few. Network representation learning methods are typically based on either a *graph factorization* or a *random-walk* based approach. The graph factorization ones (e.g., GraRep [8], TADW [48], HOPE [33]) are known to be memory intensive and computationally expensive, so they don't scale well. On the other hand, random-walk based methods (e.g., DeepWalk [36], node2vec [14]) are known to be able to scale to large networks. A comprehensive coverage of the different methods can be found in the following surveys [7] [16] [51].

A major shortcoming of these network representation learning methods is that they can only be applied on *static networks*. However, in real-world, networks are continuously evolving, as nodes and edges are added or deleted over time. As a result, any previously obtained network representation will now be outdated having an adverse effect on the accuracy of the data mining task at stake. In fact, the more significant the network topology changes are, the more likely it is for the mining task to perform poorly. One would expect though that network representation learning should account for continuous changes in the network, in an online mode. That way, (i) the low-dimensional network representation could continue being employed for downstream data mining tasks, and (ii) the results of the mining tasks obtained by the subsequent network representations would be comparable to each other. Going one step further, one would expect that while obtaining the network representation at any moment is possible, the evolving network representation learning framework suggest the best time to obtain the representation based on the upcoming changes in the network.

1.2 Research Objective

The main objective of this thesis is to develop methods for *learning representations of evolving networks*. The focus of our work is on random-walk based methods that are known to scale well. The naive approach to address this problem is to re-apply the random-walk based network representation learning method of choice every time there is an update to the network. But this approach has serious drawbacks. First, it will be very inefficient, because the embedding method is computationally expensive and it needs to run again and again. Then, the data mining results obtained by the subsequent network representations are not directly comparable to each other, due to the differences involved between the previous and the new set of random walks, as well as, the non-deterministic nature of the deep learning process itself (see chapter 3 for a detailed discussion). Therefore the naive approach would be inadequate for learning representations of evolving networks.

In contrast to the naive approach, we propose a random-walk based method for learning representations of evolving networks. The key idea of our approach is to design efficient methods that are incrementally updating the original set of random walks in such a way that it always respects the changes that occurred in the evolving network. As a result, we are able to continuously learn a new mapping function from the evolving network to a low-dimension network representation, by only updating a small number of random walks required to re-obtain the network embedding. The advantages of this approach are multi-fold. First, since the changes that occur in the network topology are typically local, only a small number of the original set of random walks will be affected, giving rise to substantial time performance gains. In addition, since the network representation will now be continuously informed, the accuracy performance of the network mining task will be improved. Furthermore, since the original set of random walks is maintained as much as possible, subsequent results of the mining tasks will be comparable to each other.

1.3 Contributions

In summary, the major contributions of this work include:

- a systematic analysis that illustrates the instability of the random-walk based network representation methods and motivates our work.
- an algorithmic framework for efficiently maintaining a valid set of random walks with respect to the changes that occur in the evolving network topology. The framework treats random walks as "documents" that are indexed using an open-source distributed indexing and searching library. Then, the index allows for efficient ad hoc querying and update of the collection of random walks in hand.

- a novel algorithm, **EvoNRL**, for Evolving Network Representation Learning based on random walks, which offers substantial time performance gains without loss of accuracy. The method is generic, so it can accommodate the needs of different domains and applications.

- an analytical method for determining the right time to obtain a new representation of the evolving network. The method is based on adaptive evaluation of the the degree of divergence between the most recent random-walk set and the random-walk set utilized in the most recent network embedding. The method is tunable so it can be adjusted to meet the accuracy/sensitivity requirement of different domains, therefore can provide support for a number of real-world applications.

- a thorough experimental evaluation on synthetic and real data sets that demonstrates the effectiveness of our method against sensible baselines, for a varying range of conditions.

An earlier version of this work appeared in the proceedings of the International Conference on Complex Networks and their Applications 2018 [17]. The conference version addressed only the case of *adding new edges*. The current version extends the problem to the cases of *deleting existing edges*, *adding new nodes* and *deleting existing nodes*. In addition, it provides an analytical method that aims to provide support to the decision making process of when to obtain a new network embedding. This decision is critical as it can effectively balance accuracy versus time performance of the method extending its applicability in domains of diverse sensitivity. In addition, it provides further experiments for the additional cases that offer substantial, new insights of the problem's complexity and the performance of our EvoNRL method.

1.4 Thesis Organization

The remainder of this thesis is organized as follows: After reviewing the related work in the following chapter 2, chapter 3 provides background and motivates our problem. Chapter 4 presents our algorithmic framework for efficiently indexing and maintaining a valid set of random walks. Our evolving network representation method and analytical method for obtaining new representations of the evolving network are presented in chapter 5. Chapter 6 presents the experimental evaluation of our methods and in chapter 7, we conclude our work.

2 Related Work

The recent success of neural networks has resulted in a significant progress in pattern recognition and data mining. Many machine learning tasks such as object detection, machine translation, and speech recognition, which once relied on feature engineering, has been recently revolutionized by end-to-end deep learning methods, i.e., convolutional neural networks (CNNs) [24], long short-term memory (LSTM) [18], and auto-encoders [47]. While deep learning methods has been particularly successfull in dealing with data which there is an underlying Euclidean structure, recently there has been a growing interest in trying to apply learning on non-Euclidean data, i.e., graph data. To handle the complexity of graph data, new generalization and methods has been rapidly developed. As a generalization of standard $2D$ convolution, graph neural networks operate on graphs. In $2D$ convolution, analogous to the graph, each pixel in an image is taken as a node where neighbours are determined by filter size. Similarly, graph convolution takes the average value of node features of the node along with its neighbours. Different from image data, the neighbours of a node are unordered and variable in size. While graph neural networks have proven to be successful in the downstream task, they are not scalable and perform better in supervised data-mining. On the other hand, random-walk based methods, inspired by the word2vec's skip-gram model of producing word embeddings [29], try to establish an analogy between *a network* and *a document*. While a document is an ordered sequence of words, a network can be effectively described by a set of random walks. It is important to note that there are many possible sampling strategies for nodes, resulting in different learned feature representations. Different strategies work better for specific prediction tasks. The methods we present are orthogonal to what features the random walks aim to learn, therefore they can accommodate most of the existing random-walk based network representation learning methods. In fact, we just need to continuously maintain a set of random walks. Our work is mostly related to research in the area of *static network representations learning* and *dynamic network representation learning*. It is also related to research in *random walks*.

2.1 Static Network Representations Learning

Network embedding aims to represent network vertices into a low-dimensional vector space, by preserving both network topology structure and node content information, so that any subsequent graph analytic tasks such as classification can be easily performed by using simple off-the-shelf machine learning algorithm. Starting with DeepWalk [36], these methods use finite length random walks as their sampling strategy and inspired by word2vec [29] use skip-gram model to maximize likelihood of observing a node's neighborhood given its low dimensional vector. This neighborhood is based on random walks. LINE [44] proposes a breadth-first sampling strategy which captures first-order proximity of nodes. In [14], authors presented *node2vec* that combines LINE and DeepWalk as it provides a flexible control of random walk sampling strategy. HARP [10] extends random walks by performing them in a repeated hierarchical manner. Also there have been further extensions to the random walk embeddings by generalizing either the embeddings or random walks [9] [37]. Role2Vec [2] maps nodes to their type-functions and generalizes other random walk based embeddings.

2.2 Dynamic Network Representation Learning

Existing work on embedding dynamic networks often apply static embedding to each snapshot of the network and then rotationally align the static embedding across each timestamp [15]. Graph factorization approaches attempted to learn the embedding of dynamic graphs by explicitly smoothing over consecutive snapshots [1]. DANE [26] is a dynamic attributed network representation framework which first proposes an offline embedding method, then updates the embedding results based on the changes in the attributed evolving network. Know-Evolve [45] proposes an evolving network embedding method in a knowledge-graph for entity embeddings based on multivariate event detection. CTDN [32] is a random walk-based continuous-time dynamic network embedding. Authors propose a temporal random walk on the network and the adjacency of nodes in the random walks refers to their adjacency in the network and in time both. The main drawback of this method is that you have to collect all snapshots of the network and then perform temporal random walks on them and it is not an online representation learning method.HTNE [52] tries to model the temporal network as a self-excited system and using Hawkes process model neighbourhood formation in the network and optimize the embedding based on point-time process. HTNE is an online dynamic network embedding framework. It uses history in its optimization and it needs to be tuned for history in each step. Moreover it uses an attention mechanism to discriminate effect of different neighbours on the target node. NetWalk [50] is a random walk based clique embedding. First in NetWalk, the reservoir is in memory which finds the next step based on the reservoir and it doesn't use any sampling method or inverted-indexing tools. In [12], authors propose a dynamic skip-gram framework. A time parameter has been added to skip-gram main optimization equation based on the application different time-stamped sentences can be fed into the neural network and the representation will respect their time. [42] proposes a dynamic word embedding which uses Gaussian random walks to project the vector representations of words over time.

2.3 Random Walks

Our work is also related to general concept of random walks on networks [27] and its applications [11, 34]. READS [20] is an indexing scheme for Simrank computation in dynamic graphs which kepng an online set of reverse-random walks and re-simulates the walks on all of the instances of the node queries. Our proposed method, kepng a set of finite-length random walks which is different from pagerank random walks and has a different sampling strategy and application compared to READS. Another aspects of random walk used in streaming data are continuous-time random walks [21]. CTRW are widely studied in time-series analysis and has applications in Finance [35]. CTRW is orthogonal to our work as we are not using time-variant random walks and our random walks do not jump over time.

3 Background and Motivation

As mentioned earlier, there are many different approaches for static network embedding. A family of these methods is based on performing random walks on a network. Random-walk based methods, inspired by the word2vec's skip-gram model of producing word embeddings [29], try to establish an analogy between *a network* and *a document*. While a document is an ordered sequence of words, a network can effectively be described by a set of random walks (i.e., ordered sequences of nodes). Typical examples of these algorithms include DeepWalk [36] and node2vec [14]. In fact, the latter can be seen as a generalization of the former, as node2vec can be configured to behave as DeepWalk. We collectively refer to these methods as **StaticNRL** for the rest of the manuscript. A typical StaticNRL method, is operating in two stpng:

(i) a set of random walks, say *walks*, is collected by performing r random walks of length l starting at each node in the network (typical values are $r = 10$, $l = 80$).

(ii) *walks* are provided as input to an optimization problem that is solved using variants of Stochastic Gradient Descent using a deep neural network architecture [5]. The context size employed in the deep learning phase is k (typical value is $k = 5$). The outcome is a set of d-dimensional representations, one for each node.

These representations are learned in an unsupervised way and can be employed for a number of predictive tasks. It is important to note that there are many possible strategies for performing random walks on nodes of a network, resulting in different learned feature representations and different strategies might work better for specific prediction tasks. The evolving network representation learning framework is orthogonal to what features the random walks aim to learn, therefore they can accommodate most of the existing random-walk based network representation learning methods.

3.1 Evaluation of the Stability of StaticNRL Methods

In this paragraph, we present a systematic evaluation of the stability of the StaticNRL methods, similar to the one presented in [3]. The evaluation aims to motivate our approach to address the problem of interest. Intuitively, a *stable* embedding method is one in which successive runs of it on the same network would learn the same (or similar) embedding. Our interest for such an evaluation is stemming from the fact that StaticNRL methods are to a great degree dependent on two random processes: (i) the set of random walks collected, and (ii) the initialization of the parameters of the optimization method. Both factors can be a source of instability for the StaticNRL method.

Comparing two embeddings can happen either by measuring their similarity or by measuring their distance. Let us introduce the following measures of instability:

- *Cosine Similarity*: Cosine similarity is a popular similarity measure for real-valued vector space models. It can also been used to compare two network embeddings using the pairwise cosine similarity on the learned d-dimensional representations [15, 22]. Formally, given the vector representations n_i and n_i' of the same node n_i in two different network embeddings obtained at two different attempts, their cosine similarity is represented as:

$$sim(n_i, n_i') = cos(\theta) = \frac{\mathbf{n_i} \cdot \mathbf{n_i'}}{||\mathbf{n_i}||\,||\mathbf{n_i'}||}$$

We can extend the similarity to two network embeddings E and E' by summing and normalizing over all nodes:

$$sim(E, E') = \frac{\sum_{i \in V} sim(n_i, n_i')}{|V|}$$

- *Matrix Distance*: Another possible way is to obtain the distance between two network embeddings by subtracting the matrices that represent the embeddings of all nodes, similarly to the approach followed in [13]. Formally, given a graph $G = (V, E)$, a network embedding is a mapping $f : V \rightarrow \mathbb{R}^d$, where $d \ll |V|$. Let $F(V) \in \mathbb{R}^{|V| \times d}$ be the matrix of all node representations. Then, we can define the following distance measure for the two network embeddings E, E':

$$distance(E, E') = ||F'(V) - F(V)||_F$$

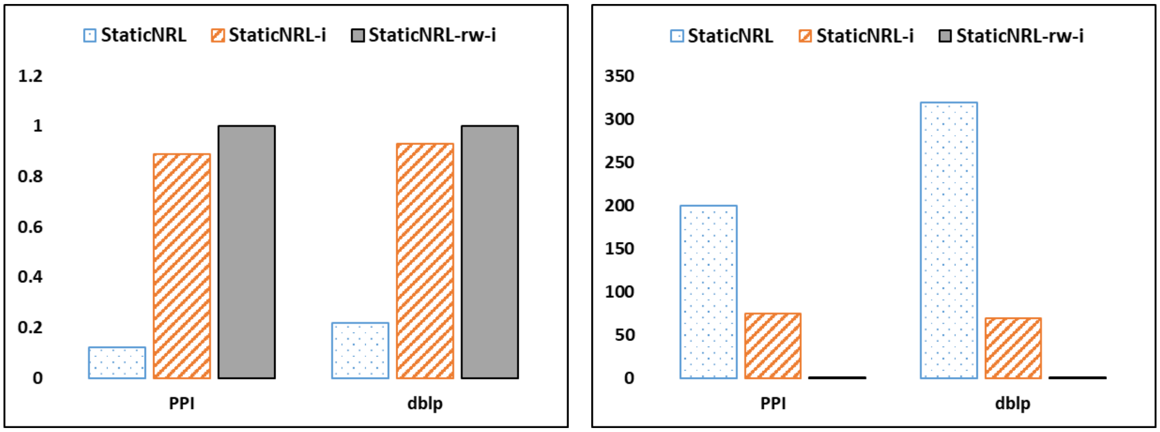
3.2 Experimental Scenario

We design a controlled experiment on two real-world networks, namely Protein-Protein-Interaction (PPI) [6] and a collaboration network (dblp) [49] that aims to evaluate the effect of the two random processes in the final network embeddings. In these experiments, we have three settings. For each setting, we run StaticNRL on a network (using parameter values: $r = 10$, $l = 10$, $k = 5$) two consecutive times, say t and $t+1$, and compute the *cosine similarity* and the *matrix distance* of the two network embeddings E^t, E^{t+1} obtained. We repeat the experiment 10 times and report averages. The three settings are:

- *StaticNRL* Each run collects independent random walks and random weights are used in the initialization phase.
- *StaticNRL-i* Each run collects independent random walks but employs the same set of weights for the initialization phase, over all runs. The purpose is to eliminate one of the random processes.
- *StaticNRL-rw-i* Each run employs the same set of random walks and the same set of weights for the initialization phase, over all runs. The purpose is to eliminate both random processes.

3.3 Results

The results of the experiment are shown in Fig. 3.1a (cosine similarity) and Fig. 3.1b (matrix distance). They show that the set of random walks and the randomized initialization



(a) cosine similarity

(b) matrix distance

Figure 3.1: Instability of the StaticNRL methods. Controlled experiments on running StaticNRL multiple times on the same network depict that the network representations learned are not stable, as a result of random initialization and random walks collected. When any of these random processes are fixed, then the network representations learned become more stable.

of the deep learning process have a significant role in moving the embedding despite the fact that there is no actual change in the topology of the network. As a matter of fact, when the same set of random walks and the same initialization is used then consecutive runs of StaticNRL result in the same embedding (as depicted by the $sim(\cdot,\cdot) = 1$ in Fig 3.1a or $distance(\cdot,\cdot) = 0$ in Fig. 3.1b). However, when the set of random walks is independent or both the random walks and the initialization are independent then substantial differences are illustrated in consecutive runs of the StaticNRL methods.

3.4 Implications

Let us start by noting that the implications of the experiment is not that StaticNRL is not useful. In fact, it has been shown to work very well. The problem is that while each independent embedding is inherently correct and has approximately same performance in downstream data mining task, these embeddings are not directly comparable to each other. In reality, the embeddings will be approximately equivalent if we are able to rotationally align them — most of similar work in the literature correct this problem by applying an *alignment method* [15]. While alignment methods can bring independent embeddings closer and eliminate the effect of different embeddings, this approach won't work well in random walk based models. The main reason for that is that as we have showed in the experiment, consecutive runs suffer from instability that is introduced by the random processes. Therefore, in the case of evolving networks (which is the focus of this work), changes that occur in the network topology will not be easily interpretable in the changes observed in the network embedding (since differences might incorporate changes due to the two random processes). However, changes in the evolving network need to be proportional to the changes in the learned network representation. For instance, minor changes in the network topology should cause small changes in the representation, and significant changes in the network topology should cause large changes in the network representation.

4 Algorithmic Framework of Dynamic Random Walks

In chapter 3, we have established the instability of random-walk based methods even when they are repeatedly applied to the same static network. That observation alone highlights the main challenge of employing these methods for learning representations of evolving networks. In this chapter we describe a general algorithmic framework and novel methods for incrementally updating the set of random walks obtained on the original network $G_t(V_t, E_t)$ at time t so that they remain valid to the updated network $G_{t'}(V_{t'}, E_{t'})$ at time t', where $t' > t$. This is the updated set of random walks could have been obtained by performing random walks on the updated network. The framework we describe is generic and can be used in any random walk-based embedding method. The first part of the chapter presents algorithms for incrementally updating the set of random walks in hand, as edges and/or nodes are added to and/or deleted from the evolving network. The second part, presents an indexing mechanism that supports the efficient storage and retrieval (i.e., querying, insert, update, deletion operations) of the set of random walks that are used for learning subsequent representations of the evolving network. A summary of notations is provided in Table 4.

Table 4.1: Summary of notations used in dynamic random walk framework

Notations	Descriptions
G_t	Network at time t
V_t	Network's vertices at time t
E_t	Network's edges at time t
G_{t+1}	Network at time $t+1$
E^+	A set of the new edges
V^+	A set of the new nodes
d_t^i	Degree of $node_i$ at time t
l	Length of a random walk
l_{sim}	Length of a simulated random walk
r	Number of random walks per node
RW_t	A set of random walks at time t
$node_i$	A node $\in V_t$
e_{ij}	A new edge ($node_i$, $node_j$)
Ind_i	The position of $node_i$ in a random walk wk
$walks_i$	Walks that contain $node_i$

4.1 Incremental Update of Random Walks

Given a network $G_t = (V_t, E_t)$ at time t, we employ a standard StaticNRL method[1] to simulate random walks. This method is configured to perform r random walks per node, each of length l (default values are $r = 10$ and $l = 80$). Let RW_t be the set of random walks obtained, where $|RW_t| = |V_t| \times r$. We store the random walks in memory, using a data structure that provides random access to its elements (i.e., a 2-D `numpy matrix`[2]). In practice, each finite-length random walk is stored as a row of a matrix, and each matrix element represents a single node of the network that is traversed by a random walk.

As we monitor changes in the evolving network, there are four distinct events that need to be addressed: i) *edge addition*, ii) *edge deletion*, iii) *node addition*, and iv) *node deletion*.

---

[1]node2vec — code is available at `https://github.com/aditya-grover/node2vec`

[2]NumPy — https://www.numpy.org/

These events can affect the network topology (and the set of random walks in hand) in different ways, therefore they need to be studied separately. First, we provide details of the *edge addition* and *edge deletion* events. This will bring up the challenges that need to be addressed in updating random walks and will introduce our main methods. Then, we visit *node addition* and *node deletion* and show that they can be treated as special cases of *edge addition* and *edge deletion*, respectively.

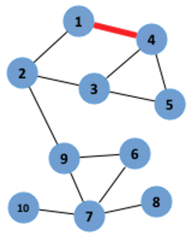
4.1.1 Edge Addition

Assume that a single new edge $e_{ij} = (node_i, node_j)$ arrives in the network at time $t+1$, so $E_{t+1} = E_t \cup (node_i, node_j)$. There are two operations that need to take place in order to properly update the set RW_t of the random walks in hand:

- *Operation 1*: contain the new edge to existing random walks in RW_t.
- *Operation 2*: discard obsolete parts of random walks of RW_t and replace them with new random walks to form the new RW_{t+1}.

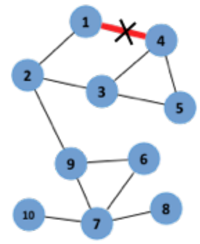
Details of each operation are provided in the next paragraphs.

Operation 1: Contain a New Edge in RW We want to update the set RW_t to contain the new edge $(node_i, node_j)$. The update should occur in a way that it represents a valid instance of a possible random walk on G_{t+1}, and at the same time, it preserves the previous set of random walks RW_t, as much as possible (to maintain network embedding stability). Note that due to the way that the original set of random walks was obtained, both $node_i$ and



RW #1 (original)	10 7 10 7 8 7 8 7 6 9
RW #1 (updated)	10 7 10 7 8 7 8 7 6 9
RW #2 (original)	4 3 2 3 5 3 5 4 3 5
RW #2 (updated)	4 3 2 3 5 3 5 4 1 2
RW #3 (original)	7 9 6 9 7 6 7 9 6 9
RW #3 (updated)	7 9 6 9 7 6 7 9 6 9
RW #4 (original)	1 2 9 7 8 7 10 7 10 7
RW #4 (updated)	1 4 3 4 3 5 4 1 4 3
RW #5 (original)	4 5 4 3 2 9 2 3 2 1
RW #5 (updated)	4 5 4 1 4 5 3 4 5 3

(a) Example addition of a new edge $(1,4)$. Random walks need to be updated to remain valid with respect to the new edge in the network. Our method guarantees that the new edge is equally represented in the updated set of random walks.



RW #1 (original)	10 7 10 7 8 7 8 7 6 9
RW #1 (updated)	10 7 10 7 8 7 8 7 6 9
RW #2 (original)	4 3 2 3 5 3 5 4 1 2
RW #2 (updated)	4 3 2 3 5 3 5 4 5 3
RW #3 (original)	7 9 6 9 7 6 7 9 6 9
RW #3 (updated)	7 9 6 9 7 6 7 9 6 9
RW #4 (original)	1 4 3 4 3 5 4 1 4 3
RW #4 (updated)	1 2 3 4 5 3 4 5 4 3
RW #5 (original)	4 5 4 3 2 9 2 9 7 10
RW #5 (updated)	4 5 4 3 2 9 2 9 7 10

(b) Example deletion of an existing edge $(1,4)$. Random walks need to be updated to remain valid with respect to the deleted edge in the network. In this example, random walk #2 and #4 traverse edge $(1,4)$ and need to be updated.

Figure 4.1: Illustrative example of EvoNRL updates for edge addition and edge deletion (colored)

$node_j$ will occur in a number of random walks of RW_t. We explain the update process for $node_i$; the same process is followed for $node_j$. First, we need to find all the random walks $walks_i \in RW_t$ that include $node_i$. Then, we need to update them so as to reflect the existence of the new edge $(node_i, node_j)$. In practice, the new edge offers a new possibility for each random walk in G_{t+1} that reaches $node_i$ to traverse $node_j$ in the next step. The number of these random walks that include $(node_i, node_j)$ depends on the node degree of $node_i$ and it is critical for correctly updating random walks in RW. Formally, if the node degree of $node_i$ in G_t is d_t then in G_{t+1} it will be incremented by one, $d_{t+1} = d_t + 1$. Effectively, a random walk that visits $node_i$ in G_{t+1} would have a probability $\frac{1}{d_{t+1}}$ to traverse $node_j$. This means that if there are $freq_i$ occurrences of $node_i$ in RW_t, then $\frac{freq_i}{d_{t+1}}$ edges $(node_i, node_j)$ need to be contained, by setting the next node of $node_i$ to be $node_j$, in the current random walk. If $node_i$ is the last node in a random walk then, there is no need to update the new edge in that random walk.

Naive approach: The naive approach to perform the updates is to visit all $freq_i$ occurrences of $node_i$ in $walks_i \in RW$ and for each of them to decide whether to perform an update of the random walk (or not), by setting the next node to be $node_j$. The decision is based on tossing a biased coin, where with probability $p_{success} = \frac{1}{d_{t+1}}$ we update the random walk, and with probability $p_{failure} = 1 - p_{success}$ we do not. While this method is accurate, it is not efficient as all occurrences of $node_i$ need to be examined, when only a portion of them needs to be updated.

Faster approach: A more efficient way is to find all the $freq_i$ occurrences of $node_i$, and then to uniformly at random sample $\frac{freq_i}{d_{t+1}}$ of them and update them by setting the next node to be $node_j$. While this method will be faster than the naive approach, it still resides on finding all the $freq_i$ occurrences of $node_i$ in the set of random walks RW, which is an expensive operation. We will soon describe how this method can be accelerated by using an efficient indexing library that allows for fast querying and retrieval of all occurrences a node in random walks.

Operation 2: Replace Obsolete Random Walks Once a new edge $(node_i, node_j)$ is contained in an existing random walk, it renders the rest of it obsolete, so it is best to be avoided. Our approach is to replace the remainder of the random walk by simulating a new random walk on the updated network G_{t+1}. The random walk starts at $node_j$ and has a length $l_{sim} = l - (Ind_i + 1)$, where $Ind_i, 0 \leq Ind_i \leq l - 1$, is the index of $node_i$ in the random walk that is currently updated. Once updates for $node_i$ have been performed, the updates that are due to $node_j$ are computed and performed.

Fig. 4.1a presents an illustrative example of how updates of random walks work, in the case of a single incoming edge on a simple network. First, a set of random walks RW_t are obtained (say 5 as illustrated by the upper lists of random walks). Let us assume that a new edge $(1,4)$ arrives. Note that now, the degree of node 1 and node 2 will increase by 1 ($d_{t+1} = d_t + 1$). Because of the new edge, some random walks need to be updated to account for the change in the topology. To perform the updates, we first search for all occurrences of i, $freq_i$. Then, we uniformly at random sample $\frac{freq_i}{d_{t+1}} = 2/2 = 1$ of them to determine where to contain the new edge. In the example, node 4 is listed after node 1 (i.e., the second node in the random walk #4 is now updated). The rest of the current random walk is obsolete, so it needs to be replaced. To perform the replacement a new random walk is simulated on the updated network G_{t+1} that starts at node 4 and has a length of $l_{sim} = l - (Ind_1 + 1) = 10 - (0 + 1) = 9$. The same process is repeated for node 4 of the added edge $(1,4)$ (see the updates in random walks #2 and #5, respectively).

The details of the proposed algorithm are described in Algorithm 1. Lines 2 and 12 of the algorithm invoke a `Query` operator. This operator is responsible for searching and retrieving information about all the occurrences of $node_i$ in the random walks set RW_t. In addition, lines 11 and 21 of the algorithm invoke a `UpdateRandomWalks` operator. This operator is responsible for updating any obsolete random walks of RW_t with the updated ones to form the new set of random walks RW_{t+1}, valid to G_{t+1}. However, these operators are very computationally expensive, especially for larger networks, and therefore will perform very poorly. In paragraph 4.2, we describe how these two slow operators,

Algorithm 1 Update RW — edge addition

procedure UPDATEWALKS
    $walks_i \leftarrow$ `Query`$(node_i)$
    $p_i \leftarrow \frac{1}{d^i_{t+1}}$
    $p_j \leftarrow \frac{1}{d^j_{t+1}}$
    $s_i \leftarrow$ `Sample`$(walks_i, p_i)$
    if $len(s_i) > 0$ **then**
        for wk in s_i **do**
            $Ind_i \leftarrow$ `Position`$(node_i, wk)$
            $l_{sim} = l - (Ind_i + 1)$
            $wk[Ind_i{+}1{:}] \leftarrow$ `SimulateWalk`$(node_j, l_{sim})$
    `UpdateRandomWalks()`
    $walks_j \leftarrow$ `Query`$(node_j)$
    $s_j \leftarrow$ `Sample`$(walks_j, p_j)$
    if $len(s_j) > 0$ **then**
        for wk in s_j **do**
            $Ind_j \leftarrow$ `Position`$(node_j, wk)$
            $l_{sim} = l - (Ind_j + 1)$
            $wk[Ind_j{+}1{:}] \leftarrow$ `SimulateWalk`$(node_i, l_{sim})$
    if $d^i_t == 0$ **then**
      RW += `SimulateWalk` $(node_i, l)$
    if $d^j_t == 0$ **then**
      RW += `SimulateWalk` $(node_j, l)$
    `UpdateRandomWalks()`

`UpdateRandomWalks` and `Query`, can be replaced by similar operators offered off-the-shelf by high performance indexing and searching open-source technologies. In addition, so far, we have relied on maintaining the set of random walks RW_t in memory. However, this is unrealistic for larger networks — while storing a network in memory as an edge list requires $O(E)$, storing the set of random walks requires $O(V \cdot r \cdot l)$ that is typically much larger for sparse networks. The indexing and searching technologies we will employ are very fast and at the same time are designed to scale to very large number of documents. Therefore, they are in position to scale well to very large number of random walks.

To accommodate a set of new edges E^+, the same algorithm needs to be applied repeatedly. The main assumption is that edges become available in a temporal order (a stream of edges), which is a common assumption for evolving networks. The premise of our method is that every time, only a small portion of the random walks need to be updated, therefore large performance gains are possible, without any loss in accuracy. In fact, the number of random walks affected depends on the node centrality of the nodes $node_i$ and $node_j$ that form the new edge $(node_i, node_j)$. While our approach suggests that a new representation is required every time a single change occurs in the network that is not the case in real-world use cases. In fact, in paragraph 5.2, we provide an analytical method for determining the right time to obtain a new representation of the evolving network. As will see the method is based on an adaptive evaluation of the the degree of divergence between the most recent random-walk set and the random-walk set utilized in the most recent network embedding. The method is tunable so it can be adjusted to meet the accuracy/sensitivity requirement of different domains, therefore can provide support for a number of real-world applications. We discuss also the implications of this issue to the time performance of the method in chapter 6.

4.1.2 Edge Deletion

Assume a single existing edge $e_{ij} = (node_i, node_j)$ is deleted from the network. Similar to edge addition, there are two operations that need to take place:

Algorithm 2 Update RW — edge deletion

```
procedure UpdateWalks
    walks ← Query(node_i, node_j)
    for wk in walks do
        Ind_i ← Position(node_i, wk)
        l_sim = l − (Ind_i + 1)
        wk[Ind_i+1:] ← SimulateWalk(node_i, l_sim)
    UpdateRandomWalks()
    walks ← Query(node_j, node_i)
    for wk in walks do
        Ind_j ← Position(node_j, wk)
        l_sim = l − (Ind_j + 1)
        wk[Ind_j+1:] ← SimulateWalk(node_j, l_sim)
    if d^i_{t+1} == 0 then                              ▷ disconnected node_i
        Remove from RW walks starting with node_i
    if d^j_{t+1} == 0 then                              ▷ disconnected node_j
        Remove from RW walks starting with node_j
    UpdateRandomWalks()
```

- *Operation 1*: delete the existing edge from current random walks in RW_t by removing any consecutive occurrence of edge's endpoints in the set.

- *Operation 2*: discard obsolete parts of random walks of RW_t and replace them with new random walks to form the new RW_{t+1}.

Details of each operation are provided in the next paragraphs.

Operation 1: Delete an Existing Edge from RW In edge deletion, unlike with the case of edge addition (where we had to sample over all the occurrences of a specific node), all the walks that have traversed the existing edge $(node_i, node_j)$ should be modified because all of them are now invalid. Other than that, the rest of the process is similar to that of edge addition. First, all random walks that have occurrences of $(node_i, node_j)$ and $(node_j, node_i)$

need to be retrieved. Then, the retrieved random walks need to be modified according to the method described in 4.1.1. Algorithm 2 describes this procedure in detail. Fig. 4.1b presents an illustrative example of updates that need to take place due to a single edge deletion. First, a set of random walks are obtained. Let us assume that a new edge $(1,4)$ is deleted, therefore random walks that traverse it, need to be updated. First, we retrieve random walks where node 1 and node 4 occur the one right after the other. For example, in random walk #4 of Figure 4.1b, node 4 appears right after 1. Since now that edge doesn't exist anymore in the network, we need to update the random walk so as to allow a valid neighbor of node 1 to appear after node 4. This action is performed in *operation 2*.

Operation 2: Replace Obsolete Random Walks This operation is similar to the one in the case of adding a new edge. We just need to replace the remainder of any random walk affected by the*Operation 1* by simulating a new random walk on the updated network G_{t+1} of the right length. Following up with the running example, to perform the replacement of the obsolete random walk, a new random walk is simulated on network G_{t+1} that starts at node 1 and has a length of $l_{sim} = l - (Ind_1 + 1) = 10 - (0+1) = 9$.

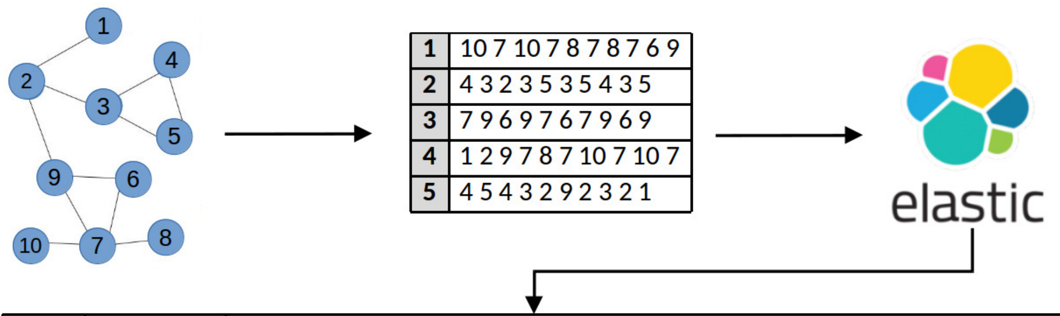
A Note About Disconnected Nodes: During the process of deleting edges, any of the edge nodes might be disconnected from the rest of the network, forming *isolated nodes*. In that case, all r random walks in RW that start from an isolated node need to be deleted. In the case that only one of the nodes of a deleted edge becomes isolated, then the simulated random walk is obtained by starting a random walk from the node that remains connected in the network.

4.1.3 Node Addition

Assume that a new node $node_i$ is added to the network at time $t+1$, so $V_{t+1} = V_t \cup \{node_i\}$. Initially, this node forms an *isolated node* (i.e., $d_i^{t+1} = 0$) and therefore there is no need to update the set of random walks RW. Now, assume that at a later time the node connects to the rest of the network through an edge $(node_i, node_j)$. In that case, we treat the new edge as described earlier in paragraph 4.1.1. In addition to that we need to simulate a set of r new random walks, each of length l, all of which start from the new node $node_i$ (recall that our original set of random walks consisted of r random walks of length l for each node in the graph). The newly obtained random walks are appended to RW^t (i.e., it is $|RW^{t+1}| = |RW^t| + r$) and are utilized in subsequent network embeddings. There is also a special case where two isolated nodes are connected. In that case we need to simulate r random walks of length l starting from each node of $node_i$ and $node_j$, respectively and append them to RW^t.

4.1.4 Node Deletion

Assume that an existing node $node_i$ is deleted from the network at time $t+1$, so $V_{t+1} = V_t - \{node_i\}$. In this case, first we obtain the set of neighbors $\mathcal{N}^i$ of $node_i$. For each $node_j \in \mathcal{N}^i$ there is an edge $(node_i, node_j)$ in the network that needs to be deleted. We delete each of these edges as described earlier in paragraph 4.1.2 and obtain the updated set RW. The deletes can occur in an arbitrary order, without any side effect. Eventually, this process forms an *isolated node*, which is removed from the graph. Deletion of the isolated



term	frequency	postings and positions
1	2	<4,1>, <5,10>
2	5	<2,3>, <4,2>, <5,5>, <5,7>, <5,9>
3	6	<2,2>, <2,4>, <2,6>, <2,9>, <5,4>, <5,8>
4	4	<2,1>, <2,7>, <5,1>, <5,3>
5	4	<2,5>, <2,7>, <2,10>, <5,2>
6	4	<1,9>, <3,3>, <3,6>, <3,9>
7	11	<1,2>, <1,4>, <1,6>, <1,8>, <3,1>, <3,5>, <3,7>, <4,4>, <4,6>, <4,8>, <4,10>
8	3	<1,5>, <1,7>, <4,5>
9	7	<1,10>, <3,2>, <3,4>, <3,8>, <3,10>, <4,3>, <5,5>
10	4	<1,1>, <1,3>, <4,7>, <4,9>

Figure 4.2: Example inverted random walk index. Given a graph, five random walks are performed. Each random walk is treated as a document and is indexed using an open-source distributed indexing and searching library. The result is an inverted index that provides information about the frequency of any node in the random walks and information about where in the random walk the node is found.

node itself doesn't further affect the set RW.

4.2 Efficient Storage and Retrieval of Random Walks

The methods of updating random walks presented in the previous paragraph are accurate. However, they depend on operators `Query` and `UpdateRandomWalks` that are computationally expensive and cannot scale to larger networks. The most expensive operation is to search the random walks RW_t to find occurrences of $node_i$ and $node_j$ of the new edge $(node_i, node_j)$. In addition, updates of random walks can be expensive as large number of existing random walks might need to be updated.

To address these shortcomings, our framework of efficiently updating random walks relies on popular open-source indexing and searching technologies. These technologies offer operations for efficiently indexing and searching large collections of documents. For example, they support efficient *full-text search* capabilities where given a query term q, all documents in the collection that contain q are retrieved. In our framework we treat each random walk as a text "document". Therefore, each node visited by a random walk would be represented as a text "term", and all random walks would represent "a collection of documents". Using this analogy, we build an *inverted random walk index*, I^{RW}. I^{RW} is an index data structure that stores a mapping from *nodes* (terms) to *random walks* (documents). The purpose of I^{RW} is to enable *fast querying* of nodes in random walks, and *fast updates* of random walks that can inform Algorithm 1. Fig. 4.2 provides an illustrative example of a small inverted random walk index. In addition, we briefly describe how to create the index and use it in our setting.

Indexing Random Walks: We obtain the initial set of random walks RW_t at time t by performing random walks on the original network, similarly to the process followed in standard StaticNRL methods. Each random walk is transformed to a document by properly concatenating the ids of the nodes in the walk. For example, a short walk $(x \rightarrow y \rightarrow z)$ over nodes x, y and z, will be represented as a document with content "x y z". These random walks are indexed to create I^{RW}. It is important to note that once an index is available, there is no need to maintain the random walks in memory any more.

Querying Random Walks: We rely on the index I^{RW} to perform any `Query` operation. Note, however, that there are additional advantages on using an efficient index. Besides searching and retrieving all random walks that contain a specific $node_i$, the index I^{RW} can be configured to provide more quantities of interest. Specifically, we configure I^{RW} so that every query retrieves additional information about the frequency of $node_i$, $freq_i$ and the position Ind_i of $node_i$ in a retrieved random walk (see Fig. 4.2). The first quantity ($freq_i$) is used to determine the number of updates that are required as discussed earlier. The second (Ind_i), is used to inform the operator `Position` in Algorithm 1 (lines 8 and 16). Note that there is a slight variation of how the `Query` operation is configured in the case of the edge deletion. Recall that in that event we need to retrieve random walks where the two nodes $node_i$ and $node_j$ are found the one right after the other (i.e., they form a step of the random walk). To accommodate this case we just need to configure the `Query` operation to retrieve all random walks that contain the bigram "$node_i$ $node_j$". A bigram is a pair of contiguous sequence of words in a document or, following the analogy, a pair of contiguous sequence of nodes in a random walk. The indexing and searching technology we employ can handily support such queries.

Updating Random Walks: We rely on the index I^{RW} for any `UpdateRandomWalk` operation. An update of a random walk is analogous to an update of a document in the index. In practice, any update of the index I^{RW} is equivalent to deleting an old random walk and then indexing a new random walk. While querying using an inverted index is a fast process, updating an index is a slower process. Therefore, the performance of our methods is dominated by the number of random walks updates required. Still, our methods would perform multitude of times faster than StaticNRL methods. A detailed analysis of this issue is provided in chapter 6. Following the discussion about the edge deletion/addition, special care is required when these events involve *isolated nodes*. In particular, if a new edge connects a previously isolated node $node_i$ to the network, then r new random walks need to be added in the index, each of which starts from $node_i$. The process of indexing the new random walks is similar to the process described in paragraph 4.2. Similarly, if an edge deletion resulted in isolating a node $node_i$, then all the r random walks that start from $node_i$ need to be removed from the index. Removing a random walk from the index is analogous to deleting a document from the index.

Bulk updates: Additional optimizations are available as a result of employing an inverted index for the random walks. For example, we can take advantage of *bulk updates*, where the index need only be updated when a number of new edges have arrived. This means that changes of single incoming edges won't be reflected in I^{RW} right away. While this optimization has the premise to make our methods faster (since updates occur once in a while), it risks harming its accuracy. In practice, it offers an interesting trade-off between accuracy and time performance that domain-specific applications need to tune. Experiments in chapter 6 demonstrate this tradeoff.

5 Evolving Network Representation Learning

So far we have described our framework for maintaining an always valid set of random walks RW_t at time t. Recall that our final objective is to be able to learn a representation of this evolving network. For the embedding process we resort to the same embedding of standard StaticNRL methods. Below we describe how embeddings of the evolving network are obtained, given a set of random walks RW_t. Then, a general strategy for obtaining an embedding only when it is mostly needed.

5.1 Learning Embeddings

Given a general network, $G_t = (V_t, E_t)$, our goal is to learn the network representation $f(V_t)$ using the skip-gram model. $f(V_t)$ is a $|V_t| \times d$ matrix where d is the network representation dimension and each row is the vector representation of a node. At the first time-stamp, the node vector representations (neural network's weights) are initialized randomly and we use this initialization for other timestamps' training. The training objective function is to maximize the log-probability of the nodes appearing in the context of the node n_i. Context of each node n_i is found using the valid RW_T set, same as the similar works in the literature [14,36]. Using the approximate objective, skip-gram with negative sampling [28], these embeddings are optimized by stochastic gradient decent so that:

$$\max_{f} \Sigma_{n_i \in V} \log Pr(n_j|\mathbf{n_i}) \tag{5.1}$$

where

$$Pr(n_j|\mathbf{n_i}) \propto \exp\left(\mathbf{n_j^T n_i}\right) \tag{5.2}$$

and $\mathbf{n_i}$ is the vector representation of a node n_i $(f(n_i) = \mathbf{n_i})$. $Pr(n_j|\mathbf{n_i})$ is the probability of the observation of neighbor node n_j, within the window-size given that the window contains n_i.

In our experiments, we use gensim implementation of skip-gram model [3]. We set our context-size, $k = 5$ and the number of dimensions, $d = 128$, unless otherwise stated.

5.2 Analytical Method for Determining the Timing of a Network Embedding

EvoNRL has the overhead of first indexing the set of initial random walks RW. At that time, we randomly initialize the skip-gram model and keep these initialization weights for the learning phase of subsequent times. As new edges/nodes are added/deleted, EvoNRL performs the necessary updates as described earlier. At each time step a valid set of random walks is available that can be used to obtain a network embedding. As we show in

[3]https://github.com/RaRe-Technologies/gensim

chapter 6 an embedding obtained by our incrementally updated set of random walks effectively represents embeddings obtained by applying a StaticNRL method directly on the updated network. However, while re-embedding the network every time a change occurs in it will result in accurate embeddings, this process is very expensive and risks to render the method non-applicable in real-world scenarios. Therefore, and depending on the domain, it is reasonable to assume that only a limited number of re-embeddings be obtained. This introduces a new problem: *when is the right time to obtain a network embedding*? In fact, this decision process demonstrates an interesting tradeoff between accuracy and time performance of the method proposed. In the rest of the paragraph we introduce two strategies for determining the time to obtain network embeddings.

PERIODIC: This is a sensible baseline where, as the name reveals, obtains embeddings periodically, every q time stpng. Depending on the sensitivity of the domain we operate on, the period can be shorter or longer. This method is easy to implement, but it is obtaining network embedding being agnostic of the different changes that occur in the network and whether they are significant (or not).

ADAPTIVE: We introduce an analytical method for determining the right timing of obtaining a network embedding. The key idea of the method is to continuously monitor the changes that occur in the network. Then, if significant changes are detected it obtains a new network embedding. In fact, we monitor two conditions, the first is able to detect occurrence of a critical change (e.g., addition of a very important edge) and is based on the idea of *peak detection*; the second is able to evaluate cumulative effects due to a number

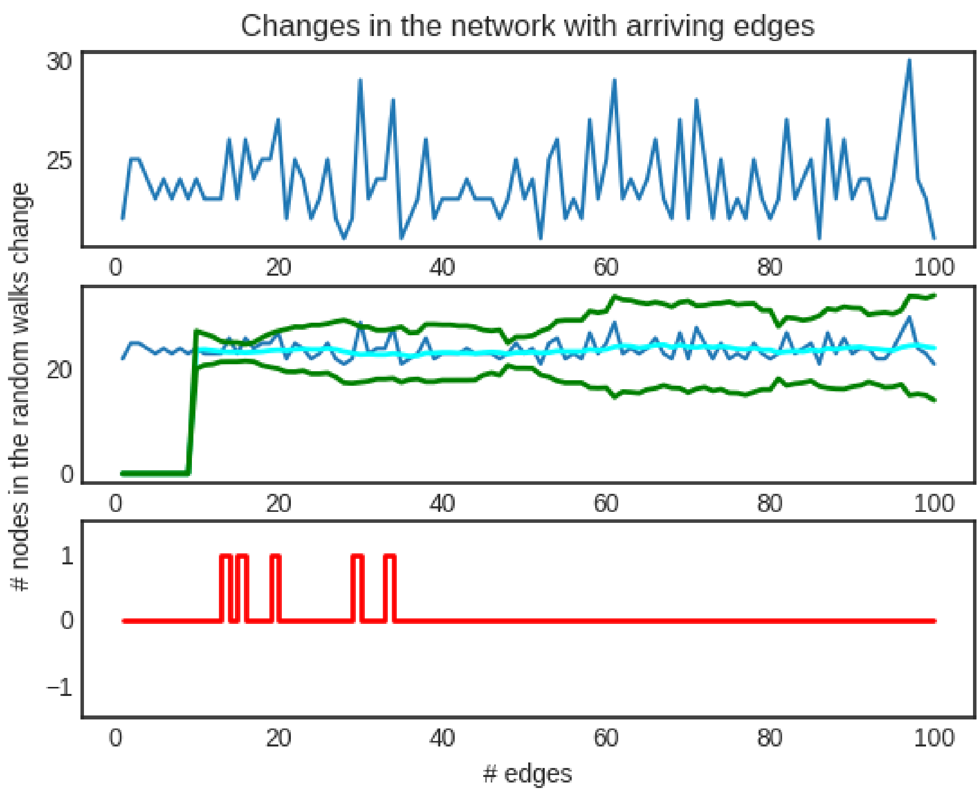


Figure 5.1: Example peak detection method for the case of adding edges in the *BlogCatalog* network. The upper plot shows the number of random walks that are updated in *RW* as a function of new edges added. It is evident that some edges have a larger effect in *RW* as depicted by higher values. The middle plot, shows the mean (middle almost straight line), as well as the boundaries defined by the current threshold of $\tau \times std$ (the two lines above and below the mean line). The bottom plot provides the signal for decision making; every time that the current change at time t is outside the threshold it signals that a network embedding should be obtained. In the example this is the case for five times $t = \{13, 15, 19, 29, 33\}$.

changes. We discuss the structure of these conditions in the following paragraphs.

Peak detection: We start by providing background of a z-score. A z-score (or standard score) is a popular statistical measure that indicates how many *standard deviations* away an observation is from its *mean*. When the population mean and the population standard

Table 5.1: Summary of notations used in decision-making algorithm

Notations	**Descriptions**
RW_t	A set of random walks at time t
RW_{t+1}	A set of random walks at time t
N_t^{t+1}	Number of the nodes changed from t to $t+1$
$\#RW_t^{t+1}$	Number of random walks changed from t to $t+1$
τ	Threshold where algorithm signals
lag	The size of the moving window
avg	Moving average of the lag window
std	Standard deviation of the lag window

deviation are unknown, the standard score may be calculated using the *sample mean* and *sample standard deviation* as estimates of the population values. In that case, the z-score of observed values x can be calculated from the following formula:

$$z = \frac{x - \hat{x}}{\hat{\sigma}} \tag{5.3}$$

where $\hat{x}$ is the mean of the sample and $\hat{\sigma}$ is the standard deviation of the sample.

In our setting, we want to detect when important changes occur in the network, so as to obtain a timely network representation. As we described earlier a good proxy for what consists an important change in a network is the number of random walks that are affected because of the change (edge addition/deletion, node addition/deletion). We can utilize the z-score of equation (5.3) to detect *peaks*. A peak or spike is a generic term which describes a sudden increase or outburst in a sequenced data [4]. In our problem, the number of random walk changes are monitored and peaks represent significant changes in the number of random walks affected. Formally, let lag be the number of changes observed in the sample. The observation window is spanning from $t - lag$ to t and we compute the mean of the sample at t as $avg[t]$. In a similar way, we calculate the standard deviation of the sample at t to be $std[t]$. Let $N[t]$ be the observation at time t that represents the number of random walks that have been updated due to a network change. Now, given $N[t]$, $avg[t]$, $std[t]$ and a threshold τ, a peak occurs at time t if the following condition holds:

$$N[t] > \tau \times std[t] + avg[t] \tag{5.4}$$

If the condition of equation (5.4) holds, then we know that a significant change has occurred and we decide to obtain a new network representation. The details of the procedure are shown in Algorithm 3. Notations used in this algorithm are summarized in Table 5.1. Figure 5.1 provides an illustrative example of the peak detection method. In this example we set $lag = 10$ and $\tau = 4$. The figure shows the results of the peak detection method for 100 changes occurring in a network (*BlogCatalog network*, *edge addition*; edges are added one by one and are randomly selected from the potential edges of the network). Our peak detection algorithm detects a total of 5 peaks occurring at $t = \{13, 15, 19, 29, 33\}$.

Cut-off score: Sometimes, changes in the network can be smooth, without any acute changes. In that case the peak detection method will fail to obtain any embedding as peaks (almost) never occur. To avoid these cases, besides the peak detection method, we employ an additional metric that monitors the cumulative effect of all the changes since the last embedding was obtained. Formally, let $N[t]$ be the observation at time t that represents the number of random walks that have been updated due to a network change. Then, the total number of random walks that have been changed between the time that the last embedding t_{old} was obtained and the current time t is given by:

$$\#RW_{t_{old}}^{t} = \sum_{t=t^{old}}^{t} N[t] \tag{5.5}$$

Now, given $\#RW_{t_{old}}^{t}$ and a threshold $cutoff$, we monitor the following condition:

$$\#RW_{t_{old}}^{t} > cutoff \tag{5.6}$$

If at any time t equation (5.6) holds, then we know that significant cumulative changes have occurred in the network and we decide to obtain a new network representation.

As we show in chapter 6 combining both conditions of equation (5.4) and (5.6) gives the best results, as it balances locally significant as well as cumulative effect of changes.

Algorithm 3 Peak Detection Algorithm

Input: *lag*, τ, *RW*
Output: *peaks*

procedure OBTAINREPRESENTATION
    `UpdateRandomWalks()`
    $N[t] \leftarrow \texttt{Length}(RW_t - (RW_t \cap RW_{t+1})$
    $avg[lag-1] \leftarrow \texttt{mean}(N[0],..,N[lag])$
    $std[lag-1] \leftarrow \texttt{std}(N[0],..,N[lag])$
    for i in $[lag+1:t]$ **do**
        if $|N[i]-avg[i-1]| < threshold * std[i-1]$ **then**
            if $N[i] > avg[i-1]$ **then**
                $peak[i] \leftarrow +1$
        else $peak[i] \leftarrow 0$
        $avg[i] \leftarrow \texttt{mean}(N[i\text{ - }lag],..,N[i])$
        $std[i] \leftarrow \texttt{std}(N[i\text{ - }lag],..,N[i])$

6 Experimental Evaluation

In this section, we experimentally evaluate the performance of our dynamic random walk framework and EVONRL[4]. In particular, we aim to answer the following questions:

- **Q1 Effect of Network Topology** How the topology of the network affects the number of random walks that need to be updated?
- **Q2 Effect of Arriving Edge Importance** How edges of different importance affect the overall random walk update time?
- **Q3 Accuracy Performance of EVONRL** What is the accuracy performance of EVONRL compared to the ground truth provided by StaticNRL methods?
- **Q4 Classification Performance of EVONRL** What is the accuracy performance of EVONRL in a downstream data-mining task?
- **Q5 Time Performance of EVONRL** What is the time performance of EVONRL?
- **Q6 Decision-Making Performance of EVONRL** How well does the strategy of EVONRL for obtaining network representations work?

[4] code is available at `https://github.com/farzana0/EvoNRL`

Q1 and **Q2** aim to shed light on the behavior of our generic computational framework for dynamically updating random walks in various settings. **Q3**, **Q4**, **Q5** and **Q6** aim to demonstrate how EVONRL performs. Before presenting the results, we provide details of the computational environment and the data sets employed.

Environment: All experiments are conducted on a workstation with 8x Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz and 64GB memory. Python 3.6 is used and the static graph calculations use the state-of-the-art algorithms for the relevant metrics provided by the *NetworkX* network library.

Data: For the needs of our experiments both synthetic data and real data sets have been employed.

- *Protein-Protein Interactions (PPI):* We use a subgraph of PPI for Homo Sapiens and use the labels from the preprocessed data used in [14]. The network consists of 3890 nodes, 76584 edges and 50 different labels.

- *BlogCatalog [41]:* BlogCatalog is a social network of blogers which each edge indicates a social interaction among them. This network consists of 10312 nodes, 333983 edges and 39 different labels.

- *Facebook Ego Network [25]:* Facebook ego network is the combined ego network of each node. There is an edge from a node to each of its friends. This network consists of 4039 nodes, 88234 edges.

- *Arxiv HEP-TH [25]:* Arxiv HEP-TH (high energy physics theory) network is the citation network from e-print Arxiv. If paper i cites paper j, there is a directed edge from i to j. This network consists of 27770 nodes, 352807 edges.

- *Synthetic Networks*: We create a set of Watts-Strogatz [31] random networks of different sizes ($n = \{1000, 10000\}$) and different rewiring probabilities ($p = \{0, 0.5, 1.0\}$). The rewiring probability is used to create representative *Lattice* ($p = 0$), *Small-world* ($p = 0.5$) and *Erdos-Reyni* ($p = 1$) networks, respectively.

6.1 Q1 Effect of Network Topology

We evaluate the effect of randomly adding a number of new edges in networks of different topologies, but same size. For each case, we report the number of the random walks that need to be updated. Fig. 6.1 shows the results, where it becomes clear that as more new edges are added, more random walks are affected. The effect is more stressed in the case of the *Small-world* and *Erdos-Reyni* networks. This is to be expected, since these networks are known to have small diameter, therefore every node is easily accessible from any other node. As a result, every node has a high chance to appear in any random walk. In contrast, *Lattices* are known to have larger diameter, therefore only a small number of nodes (out of all nodes in the network) can be accessible by any random walk. As a result, nodes are more equally distributed in all random walks.

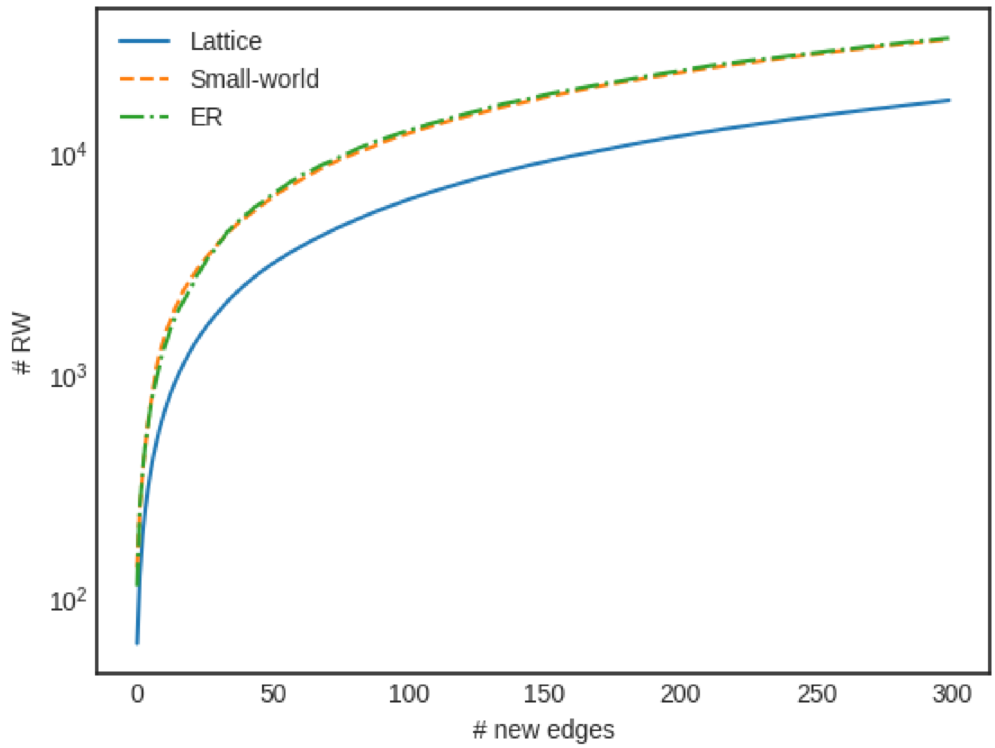
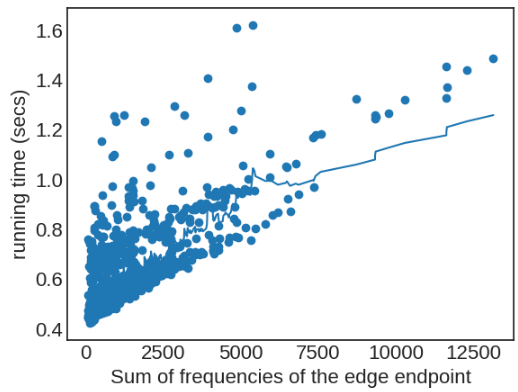


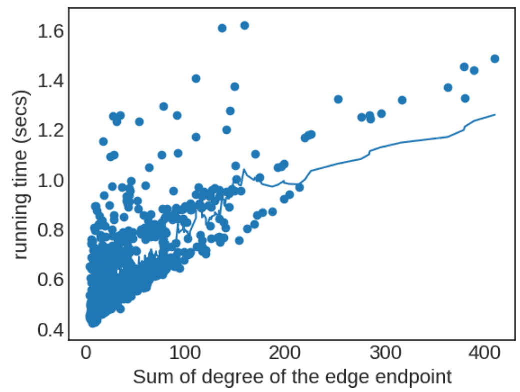
Figure 6.1: Effect of network topology (the axis of *#RW* affected is in logarithmic scale). As more new edges are added, more random walks are affected. The effect is more stressed in the case of the *Small-world* and *Erdos-Reyni* networks, than the *Lattice* network.

6.2 Q2 Effect of Arriving Edge Importance

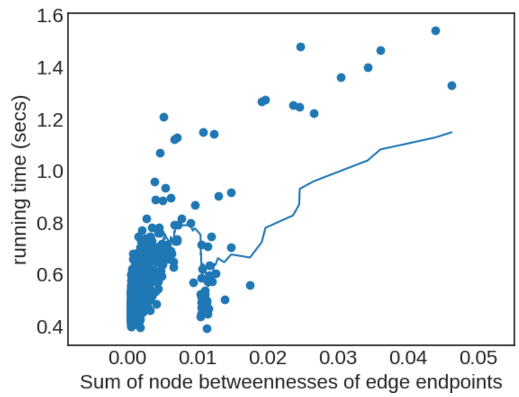
By answering **Q1**, it becomes evident that even a single new edge can have a dramatic effect in the number of random walks that need to be updated. Eventually, the number of random walks affected, will have an effect to the time performance of updating these random walks in our framework. In this set of experiments we perform a systematic analysis of the effect of the importance of an arriving edge to the time required for the update to occur. Importance of an incoming edge $e_{ij}^{t+1} = (n_i, n_j)$ at time $t+1$ in a network can



(a) frequency of the new edge endpoints



(b) node degree of the new edge endpoints



(c) node betweenness of the new edge endpoints

Figure 6.2: Dependency of EvoNRL running time on importance of added edge as described by various metrics on PPI Network.

be defined in different ways. Here, we define three metrics of edge importance, based on properties of the **endpoints** n_i, n_j of the arriving edge:

- *Sum of frequencies of edge endpoints in* RW_t.
- *Sum of the node degrees of edge endpoints in* G_t.
- *Sum of the node-betweenness of edge endpoints in* G_t.

Results of the different experiments are presented in Fig. 6.2. The first observation is that important incoming edges are more expensive to update, sometimes up to three or four times (1.6sec vs 0.4sec). This is expected, as more random walks need to be updated. However, the majority of the edges are of least importance (lower left dense areas in Fig. 6.2a, Fig. 6.2b and Fig. 6.2c), so fast updates are more common. Finally, the behavior of sum of node frequencies (Fig. 6.2a) and sum of node degrees (Fig. 6.2b) of the edge endpoints are correlated. This is because the node degree is known to be directly related to the number of random walks that traverse it. On the other hand, node-betweenness demonstrates more unstable behavior since it is mostly related to shortest paths and not just paths (which are related to random walks).

6.3 Q3 Accuracy Performance of EvoNRL

In this set of experiments we evaluate the accuracy performance of EvoNRL and show that it is very **accurate**. At this point, it is important to note that evidence of our EvoNRL performing well is provided by demonstrating it obtains **similar representations** to the ground truth provided by running StaticNRL on different instances of the evolving network. This is because the objective of our method is to resemble as much as possible what the actual changes in the original network are by incrementally maintaining a set of random walks and monitoring the changes. In practice, we aim to show that our proposed algorithm is able to update random walks in a way that they are always representing a valid instance of random walks that can be obtained by running StaticNRL on the updated network. In these experiments, we show the representation learned by EvoNRL and the ground truth provided by the StaticNRL are similar to each other by using *a representational similarity* metric.

6.3.0.1 Similarity of Two Representations

Our goal here is to compare the representations learned by the neural network and show that EvoNRL results in a similar representations to ground truth provided by StaticNRL methods. Comparing representations in neural networks is difficult as the representations vary even across the neural networks trained on the same input data with the same task [39]. In this paper, representations are weights of the representation learned by either our EvoNRL method or the StaticNRL method, and they represent the representation learned by a skip-gram neural network. In order to determine the correspondence between these representations, we use the recent similarity measures of neural networks studied in [30] and [23]. Dynamics of neural networks call for a similarity metric that is *invariant to orthogonal transformation* and *invariant to isotropic scaling*. Assuming two representations $X \in \mathbb{R}^{n \times d}$ and $Y \in \mathbb{R}^{n \times d}$, we are concerned about a scalar similarity index $s(X,Y)$ which can be used to compare the two neural network representations. There are many methods for comparing two finite set of vectors and measure the similarity between them. The simplest approach is to employ a dot-product based similarity. By summing the square dot-product of each corresponding pair of vectors in X and Y, we can have a similarity index between matrices X and Y. This approach is not practical as representations of the neural networks can be described on two different basis and result in a misleadingly similarity index. Therefore invariance to linear transforms is crucial in neural network representational similarity metrics. Recently, Canonical Correlation Analysis (CCA) [19] is used as a tool to compare representations across networks. Canonical Correlation Analysis has been widely used to evaluate the similarity between computing models and brain activity. CCA can find similarity between representations where they are superficially dissimilar. Its invariance to linear transforms makes CCA a useful tool to quantify the similarity of EvoNRL and StaticNRL representations [30].

Canonical Correlation Analysis (CCA): Canonical Correlation Analysis [19] is a statistical technique to measure the linear relationship between two multidimensional set of vectors. Ordinary Correlation analysis is highly dependent on the basis which the vectors are described on. The important property of CCA is that it is invariant to affine transformations of the variables which makes it a proper tool to measure representation's similarity by. If we have two sets of matrices $X \in \mathbb{R}^{n \times d}$ and $Y \in \mathbb{R}^{n \times d}$, Canonical Correlation Analysis will find two bases, one for X and one for Y such that after their projections into these bases, their correlation will be maximized. for $1 \leq i \leq d$, the i^{th}, canonical correlation coefficient is given by:

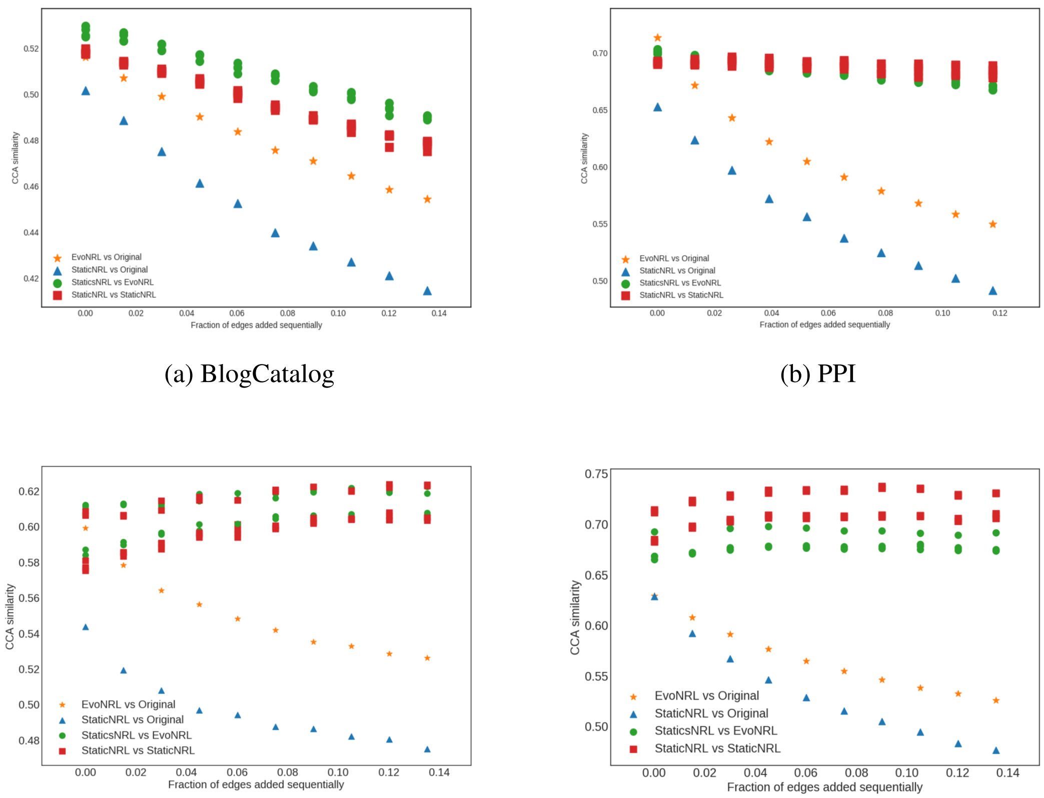
$$\begin{aligned} \rho_i = \max_{w_X^i, w_Y^i} corr(Xw_X^i, Yw_Y^i) \\ subject\ to\ \forall_{j<i} Xw_X^i \perp Xw_X^j \\ \forall_{j<i} Yw_Y^i \perp Yw_Y^j \end{aligned} \tag{6.1}$$

where the vectors $w_X^i \in \mathbb{R}^d$ and $w_Y^i \in \mathbb{R}^d$ transform the original matrices into canonical variables Xw_X^i and Yw_Y^i.

$$R^2_{CCA} = \frac{\Sigma_{i=1}^{d} \rho_i^2}{d} \tag{6.2}$$

The mean squared CCA correlation [40], R^2_{CCA} reports the sum of the squared canonical correlations. This sum is a metric that shows the similarity of the two multidimensional sets of vector.

Experimental scenario: In these experiments, the original network is the initial network at the beginning. We simulate random walks on this network and learn its representation. After that, we sequentially make changes (add edges, remove edges, add nodes and remove nodes) to the initial network and keep the random walks updated using EvoNRL. In certain points (for example after every 1000 edge addition in the PPI network), we learn the network representation in two ways. One is by simulating new random walks on the updated network (original network with new edges/nodes or missing edges/nodes) and second is learning the representation using EvoNRL. Now we have two representations of the same network and the goal is to compare them to see how similar EvoNRL is to StaticNRL. Note that StaticNRL simulates walks on the updated networks while EvoNRL has been updating the original random walk set. Representations obtained by StaticNRL are results of simulating random walks on the network. Because of the randomness involved in the process, it is typical that two differnet StaticNRL representations of the same network are not identical. We can measure, the similarity of the different representations using CCA. In our evaluation, we aim to demonstrate that EvoNRL is as similar to StaticNRL and that this similarity is comparable to the similarity obtained by applying StaticNRL multiple times



(a) BlogCatalog

(b) PPI

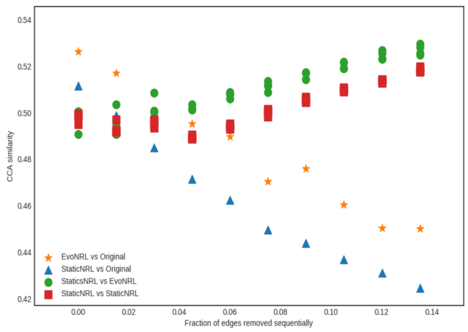
(c) Facebook

(d) Cit-HepTh

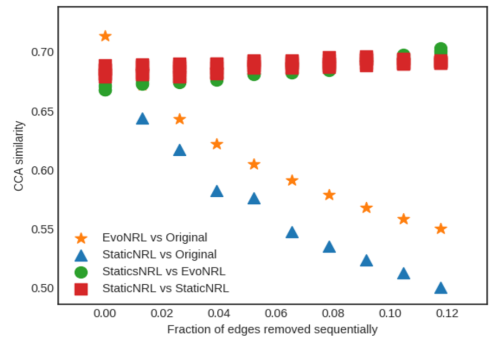
Figure 6.3: Accuracy performance of EvoNRL — adding edges.

on the same network. At any stage of the change (edge addition, edge deletion, node addition, node deletion) in the network, EvoNRL is updating the random walk set in a way that it is representing the network. First, we run StaticNRL multiple times (x5) on a network. Each StaticNRL is simulating a random walk set on the evolving network at certain times. Representations are two finite sets of vectors in d-dimensional space and then we compare how similar these two sets are.

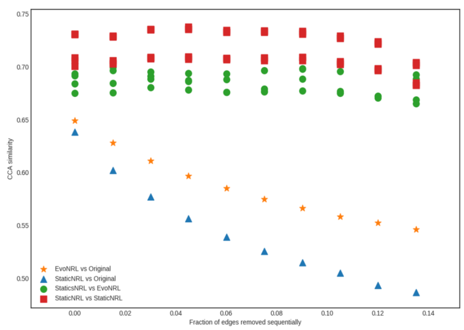
Adding edges: Given a network $G = (V, E)$, we can add a new edge by randomly picking two nodes in the network that are not currently connected and connect them. Adding new



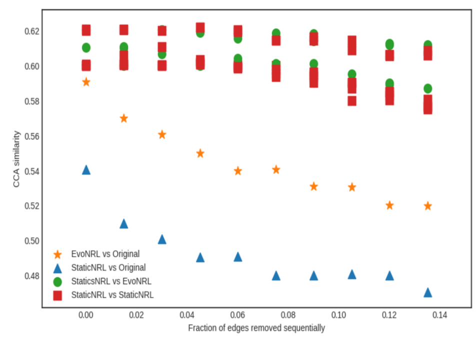
(a) BlogCatalog



(b) PPI



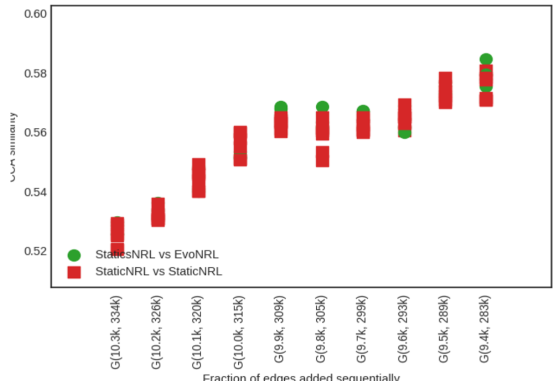
(c) Cit-HepTh



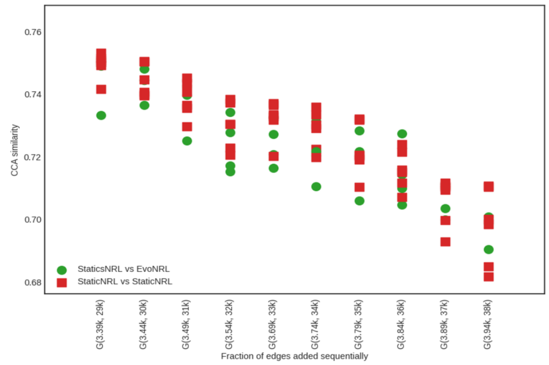
(d) Facebook

Figure 6.4: Accuracy performance of EvoNRL — removing edges.

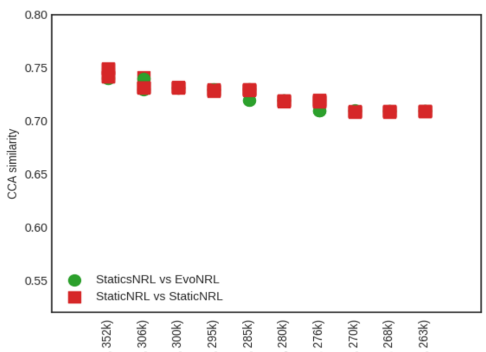
edges to the network should have an effect on the network embedding. By adding edges, as the network diverges from its original state, the embedding will diverge from the original network as well. Fig. 6.3 shows the accuracy results of EvoNRL. We observe that the CCA similarity index of EvoNRL follows the same trend as the StaticNRL in all the networks: BlogCatalog (Fig. 6.3a) and the PPI (Fig. 6.3b), Facebook (Fig. 6.3c) and Cit-HepTh (Fig. 6.3d) networks. The similarity of the two methods remains consistent as more edges are added (up to 12% of the number of edges in the original PPI; up to 14% of the number of edges in the original BlogCatalog, Facebook and Cit-HepTh). In Fig. 6.3, there are



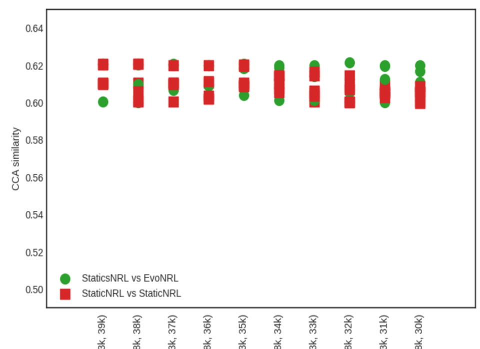
(a) BlogCatalog



(b) PPI



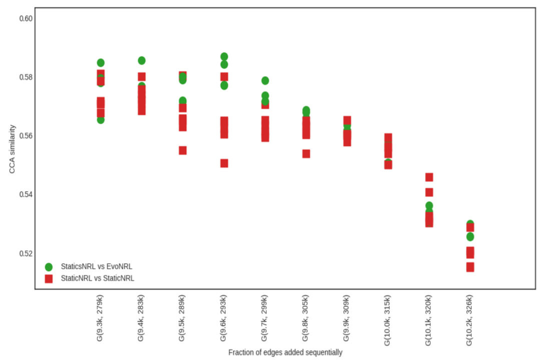
(c) Cit-HepTh



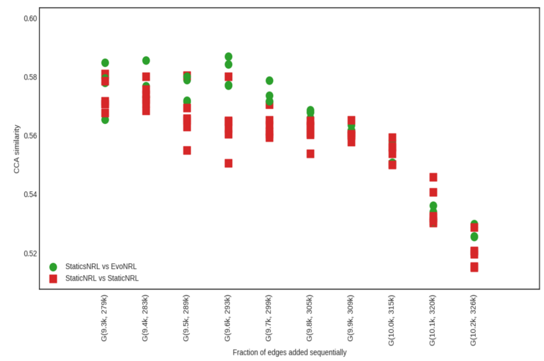
(d) Facebook

Figure 6.5: Accuracy performance of EVONRL — removing nodes.

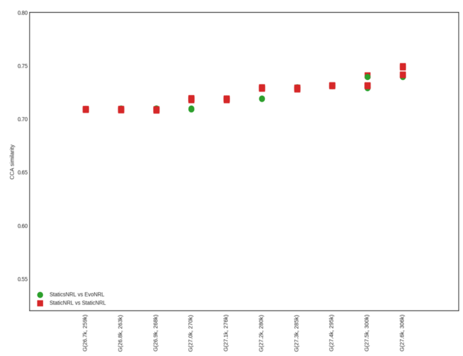
two sorts of comparison. First, The similarity of EvoNRL and the Original Network (The network before changes occur to it) is measured. The decreasing trend in orange stars in Fig. 6.3 shows that EvoNRL is updating the set of random walks and the representations of the updated networks are diverging from the representation of the original network. On the other hand, we see that EvoNRL is more correlated to the original set of the random walk (orange stars), compared to StaticNRL (Blue Triangles). Blue Triangles are the average of canonical correlation of the original network with 4 different runs of StaticNRL. It shows that the representation of the evolving network is diverging from the original network. So



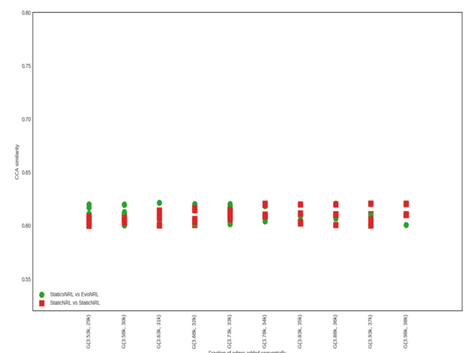
(a) BlogCatalog



(b) PPI



(c) Cit-HepTh



(d) Facebook

Figure 6.6: Accuracy performance of EvoNRL — adding nodes.

far we have showed that EvoNRL is consistently updating the original set of random walks and makes difference in the network's representation. The question is are these updates accurate? To answer this question we add edges step by step to the original network. Using EvoNRL we keep updating a set of random walk and get the representation of the network in a certain points. On the other hand, we run StaticNRL on the updated network at the same certain points. Because of the randomness of the random walks we repeat StaticNRL for 4 times. We compare the StaticNRL representations obtained from the same network with each other to have a baseline of the similarity metric. The red squares showing as

’StaticNRL vs StaticNRL’ in Fig. 6.3 are showing the average similarity of representations of StaticNRL compared to each other 2 by 2. Our goal is to show, EvoNRL keeps updating the random walk set in an accurate way and the representation obtained by EvoNRL is as accurate as StaticNRL. To show this, we measure the canonical correlation of EvoNRL representation and the StaticNRl. We observe that (green circles) EvoNRL representations is very similar to the StaticNRL representations and can be an instance on StaticNRL.

Removing edges: Given a network $G = (V, E)$, we can remove an edge by randomly choosing an existing edge $e \in E$ and remove it from the network. Removing existent edges should have an effect in the network embedding. Fig. 6.4 show the accuracy results of edge deletion. Similar to edge addition, We observe that the CCA similarity of EVONRL follows the same trend as the StaticNRL in all the networks: BlogCatalog (Fig. 6.4a) and the PPI (Fig. 6.4b), Facebook (Fig. 6.4c) and Cit-HepTh (Fig. 6.4d) networks.

Adding nodes: As we described in Section 5 node addition can be treated as a special case of edge addition. This is because whenever a node is added in a network, a number of edges attached to that node need to be added as well. To emulate this process, given a network $G = (V, E)$, first we create a network $G' = (V', E')$, where $V' \subseteq V, E' \subseteq E$ as follows. We uniformly at random sample nodes $V' \subseteq V$ from G and then remove these nodes and all their attached edges $E' \subseteq E$ from G, forming G'. Following that process, we obtain a new network for BlogCatalog with $V' = 8312$ and a new network for PPI with $V' = 3390$ nodes, respectively. Then, we start adding the nodes $v \in V'' = V \setminus V'$ that have been removed from G, one by one. Whenever, a node $v \in V''$ is added to G', any edge between v and nodes existing in the current state of network G' are added as well. Adding nodes to the network should have an effect in the network embedding. Fig. 6.6 shows the accuracy results of node addition. CCA compares two sets of vectors with the same cardinality. Because the number of the nodes and therefore the number of the vectors in the representation are variant, we can not compare the updated representations with the original network. In these experiments we show that EvoNRL and StaticNRL on the same network are very similar to each other and EvoNRL is an accurate instance of StaticNRL.

Removing nodes: As we described in Section 5 node deletion can be treated as a special case of edge deletion. Given a network $G = (V, E)$, we start removing nodes $v \in V$ from the network, one by one. When a node is removed all the edges connecting this node to the network are removed as well. The process of removing nodes will result in a new network $G'(V', E')$, where $V' \subseteq V$ and $E' \subseteq E$. Removing existing nodes from the network effect in the network embedding. Fig. 6.5 shows the accuracy result of node deletion. In the evolving network, nodes are removed from the network sequentially and EvoNRL keeps updating a valid set of random walks. we show that the representations obtained from these random walks are similar to StaticNRL representations. Same as node addition, because the number of the nodes are changing, we can not compare the representations with the original network's representation. The experiments above provides strong evidence that our random walk updates are correct and can incrementally maintain a set of random walks that is their corresponding representations are similar to that of obtained by StaticNRL.

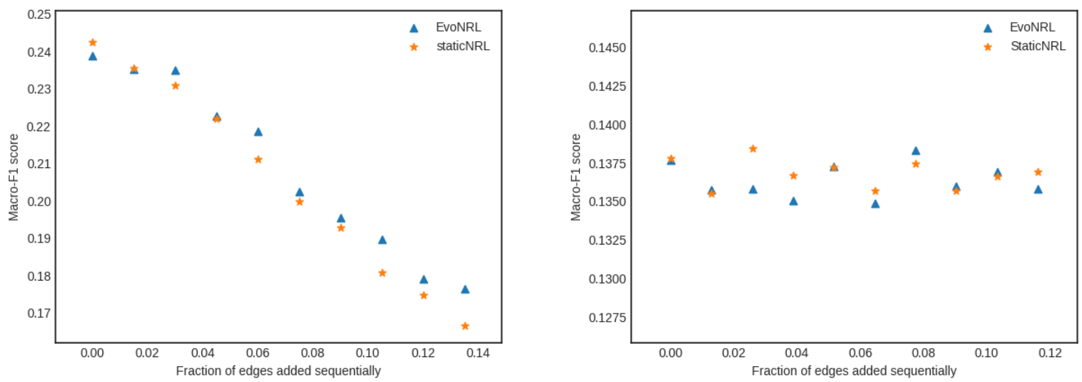
6.4 Q4 Classification Performance of EvoNRL

In this set of experiments we evaluate the accuracy performance of EvoNRL and show that it is very **accurate**. At this point, it is important to note that evidence of our EvoNRL performing well is provided by demonstrating it has **similar accuracy** to StaticNRL, for the various aspects of the evaluation (and **not** by demonstrating loss/gains in accuracy). This is because the objective of our method is to resemble as much as possible what the actual changes in the original network are by incrementally maintaining a set of random walks and monitoring the changes. In practice, we aim to show that our proposed algorithm is able to update random walks in a way that they are always representing a valid instance of random walks that can be obtained by running StaticNRL on the updated network.

Experimental scenario: To evaluate our random walk update algorithm, we resort to accuracy experiments performed on a downstream data mining task: *multi-label classification*. The network topology of many real-world networks can change over time due to either adding/removing edges or adding/removing nodes in the network. In our experimental scenario, given a network we simulate and monitor network topology changes. Then, we run StaticNRL multiple times, one time after each network change and learn multiple network representations over time. The same process is followed for EvoNRL but this time we only need to update the random walks RW_t at each time t and use these for learning multiple network representations over time. In multi-label classification each node has one or more labels from a finite set of labels. In our experiments, we see 50% of nodes and their labels in the training phase and the goal is to predict labels of the rest of the nodes. We use node vector representations as input to a one-vs-rest logistic regression classifier with L2 regularization. Finally, we report the $Macro - F_1$ accuracy of the multi-label classification of StaticNRL and EVONRL as a function of the fraction of the network changes. For StaticNRL, since it is sensitive to the fresh set of random walks obtained every time, we run multiple times (10x) and report the averages. We experiment with the *BlogCatalog* and *PPI* networks. In the following paragraphs we present and discuss the results for each of the interesting cases (adding/removing edges, adding/removing nodes).

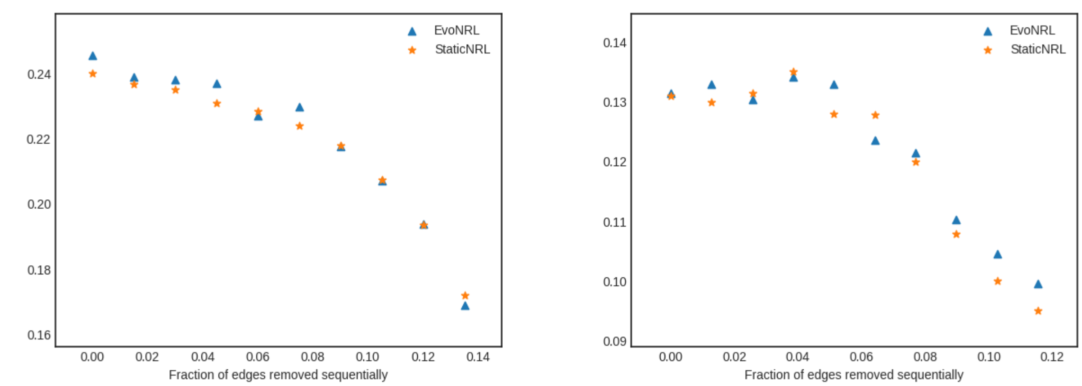
Adding edges: Given a network $G = (V, E)$, we can add a new edge by randomly picking two nodes in the network that are not currently connected and connect them. Adding new edges to the network should have an effect on the network embedding and thus in the overall accuracy of the classification results. Fig. **??** shows the results. We observe that the Macro-F_1 accuracy of EVONRL follows the same trend as the one of StaticNRL in both the BlogCatalog (Fig. 6.7a) and the PPI (Fig. 6.7b) networks. The accuracy of the two methods remains consistent as more edges are added (up to 12% of the number of edges in the original PPI; up to 14% of the number of edges in the original BlogCatalog). This provides strong evidence that our random walk updates are correct and can incrementally maintain a set of random walks that is similar to that obtained by StaticNRL when applied in an updated network.

Removing edges: Given a network $G = (V, E)$, we can remove an edge by randomly choosing an existing edge $e \in E$ and remove it from the network. Removing existent edges should have an effect in the network embedding and thus in the overall accuracy of the classifica-



(a) BlogCatalog (b) PPI

Figure 6.7: Accuracy performance of EvoNRL — adding new edges.



(a) BlogCatalog (b) PPI

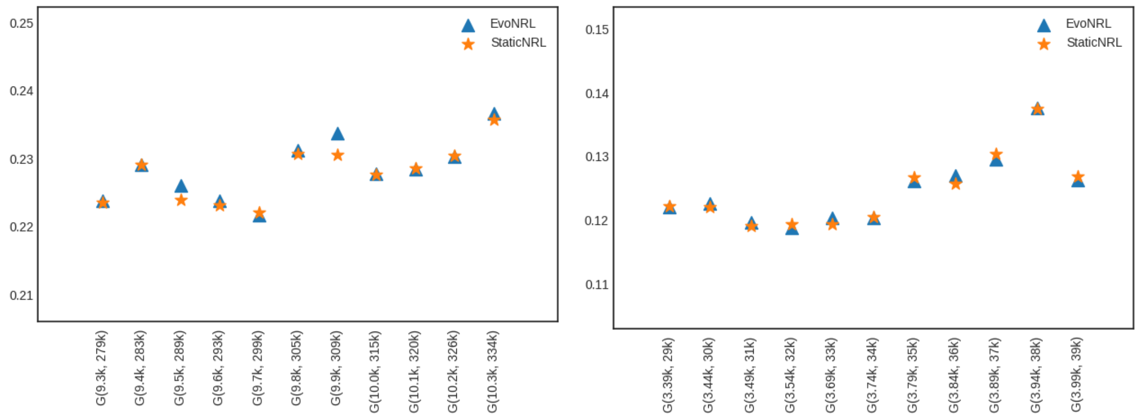
Figure 6.8: Accuracy performance of EvoNRL — removing edges.

tion results. We evaluate the random walk update algorithm for the case of edge deletion in a way similar to that of adding edges. The only difference is that every time an edge is deleted at t we update random walks to obtain RW_t. Then, the updated RW_t can be used for obtaining a network representation. Same setting is used in multi-label classification. Fig. 6.8 shows the results. Again we observe that the Macro-F$_1$ accuracy of EvoNRL follows the same trend as the one of StaticNRL in both the BlogCatalog (Fig. 6.8a) and the PPI

(Fig. 6.8b) networks.

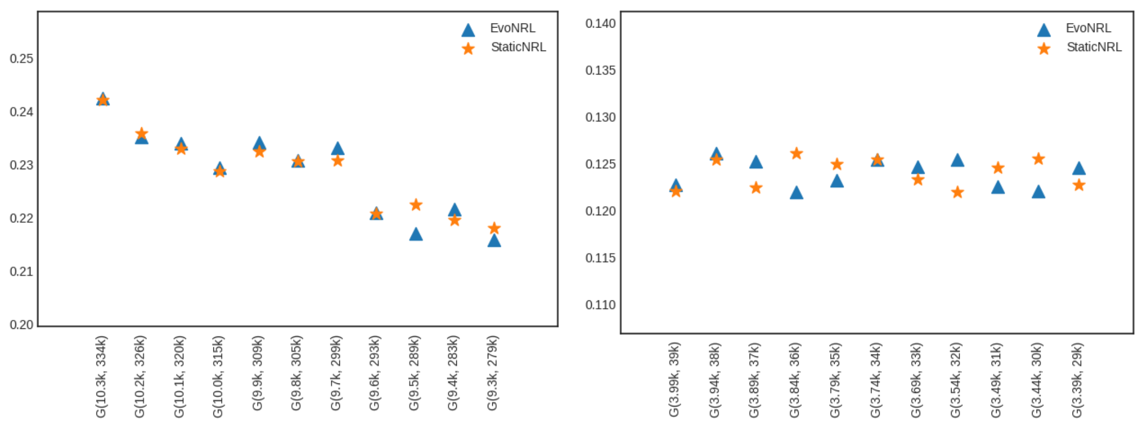
Adding nodes: As we described in Section 5 node addition can be treated as a special case of edge addition. This is because whenever a node is added in a network, a number of edges attached to that node need to be added as well. To emulate this process, given a network $G = (V,E)$, first we create a network $G' = (V',E')$, where $V' \subseteq V, E' \subseteq E$ as follows. We uniformly at random sample nodes $V' \subseteq V$ from G and then remove these nodes and all their attached edges $E' \subseteq E$ from G, forming G'. Following that process, we obtain a new network for BlogCatalog with $V' = 8312$ and a new network for PPI with $V' = 3390$ nodes, respectively. Then, we start adding the nodes $v \in V'' = V \setminus V'$ that have been removed from G, one by one. Whenever, a node $v \in V''$ is added to G', any edge between v and nodes existing in the current state of network G' are added as well. Adding nodes to the network should have an effect in the network embedding and thus in the overall accuracy of the classification results. We evaluate the random walk update algorithm for the case of node addition in a way similar to that of adding edges. The only difference is that every time a node is added at t we update random walks to obtain RW_t, by adding a number of edges. Then, the updated RW_t can be used for obtaining a network representation. Fig. 6.9 shows the results. Again we observe that the Macro-F_1 accuracy of EVONRL follows the same trend as the one of StaticNRL in both the BlogCatalog (Fig. 6.9a) and the PPI (Fig. 6.9b) networks.

Removing nodes: As we described in Section 5 node deletion can be treated as a special case of edge deletion. Given a network $G = (V,E)$, we start removing nodes $v \in V$ from



(a) BlogCatalog (b) PPI

Figure 6.9: Accuracy performance of EvoNRL — adding new nodes.



(a) BlogCatalog (b) PPI

Figure 6.10: Accuracy performance of EvoNRL — removing new nodes.

the network, one by one. When a node is removed all the edges connecting this node to the network are removed as well. The process of removing nodes will result in a new network $G'(V', E')$, where $V' \subseteq V$ and $E' \subseteq E$. Removing existing nodes from the network should have an effect in the network embedding and thus in the overall accuracy of the classification results. We evaluate the random walk update algorithm for the case of node deletion in a way similar to that of deleting edges. The only difference is that every time a node is deleted at t we update random walks to obtain RW_t, by removing a number of

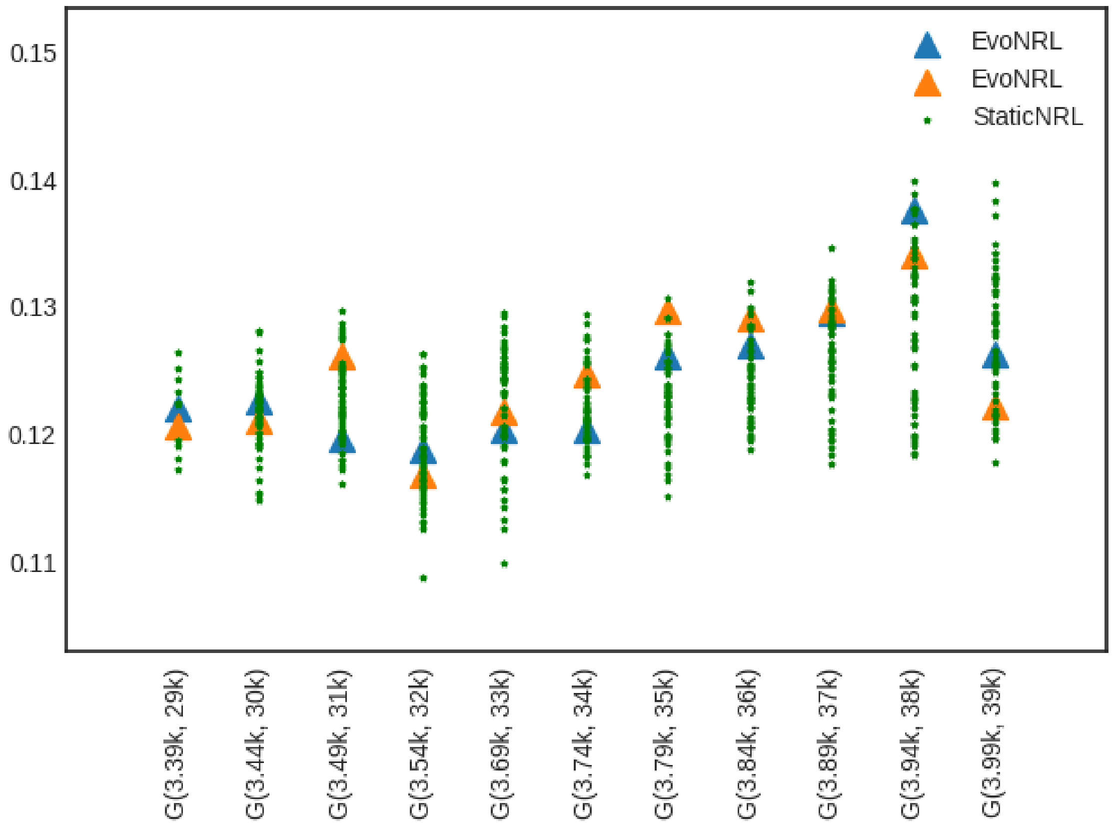


Figure 6.11: Accuracy values obtained by running StaticNRL multiple times on the same network. The values are significantly fluctuating due to sensitivity to the set of random walks obtained. Similarly, EVONRL is sensitive to the initial set of random walks obtained. Two instances of EVONRL are shown, each of which operates on a different initial set of random walks.

edges. Then, the updated RW_t can be used for obtaining a network representation. Fig. 6.10 shows the results. Again we observe that the Macro-F_1 accuracy of EVONRL follows the same trend as the one of StaticNRL in both the BlogCatalog (Fig. 6.10a) and the PPI (Fig. 6.10b) networks.

Discussion about accuracy value fluctuations: While we have demonstrated that EVONRL is able to resemble the accuracy performance obtained by StaticNRL, one can observe that in some cases the accuracy values of the methods can substantially fluctuate. This behavior can be explained by the sensitivity of the StaticNRL methods to the set of random walks obtained from the network, as discussed in the motivating example of Section 3.1. EVONRL would also inherit this problem, as it depends on an initially obtained set of random walks that is subsequently updated at every network topology change. To demonstrate this sensitivity effect, we run control experiments on the PPI network for the case of adding new nodes in the network G, similar to the experiment in Fig. 6.9b. However, this time, instead of reporting the average over a number of runs for the StaticNRL method, we report all its instances. In particular, as we add more nodes (the number of nodes increases from 3390 to 3990) a new network is obtained. We report the accuracy values obtained by running StaticNRL multiple times (40x) on the same network. We also depict the values of two different runs for EVONRL. Each run obtains an initial set of random walks that is incrementally updated in subsequent network topology changes. It becomes evident that the StaticNRL values can significantly fluctuate due to the sensitivity to the set of random walks obtained. It is important to note that EVONRL manages to fall within the range of these fluctuations.

6.5 Q5 Time Performance of EVONRL

In this set of experiments we evaluate the time performance of our method and show that EVONRL is very **fast**. We run experiments on two *Small-world* networks (Watts-Strogatz ($p = 0.5$)), with two different number of nodes ($|V| = 1000$ and $|V| = 10000$). We evaluate

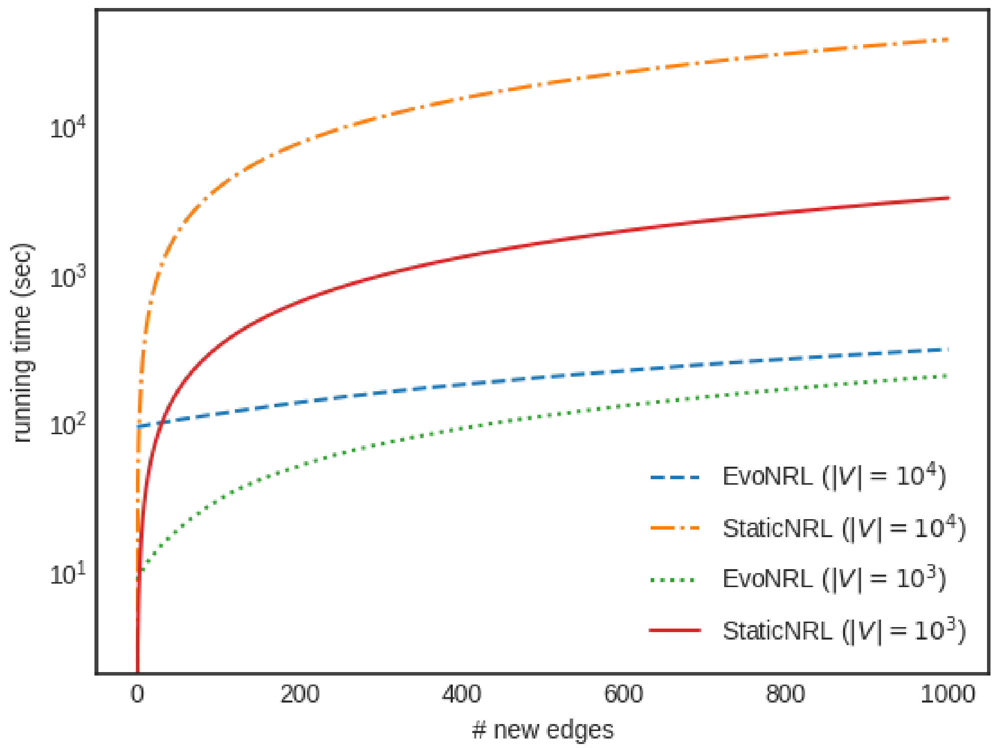


Figure 6.12: EVONRL scalability (running time axis is in logarithmic scale). StaticNRL scales linearly to the number of new edges added in the network, since it has to run again and again for every new edge. At the same time, EVONRL is able to accommodate the changes more than 100 times faster than StaticNRL. This behavior is even more stressed in the larger network (where the number of nodes is larger).

EVONRL against a standard StaticNRL method from the literature [14]. Both algorithms start with the same set of random walks RW. As new edges are arriving, StaticNRL needs to learn a new network representation by resimulating a new set of walks every time. On the other hand, EVONRL has the overhead of first indexing the set of initial random walks RW. Then, for every new edge that is arriving it just needs to perform the necessary updates as described earlier. Fig. 6.12 shows the results. It can be seen that the performance of StaticNRL is linear to the number of new edges, since it has to run again and again for every new edge. At the same time, EVONRL is able to accommodate the changes more than 100 times faster than StaticNRL. This behavior is even more stressed in the larger network (where the number of nodes is larger). By increasing the number of nodes, running StaticNRL becomes significantly slower, because by design it needs to simulate larger amount of random walks. On the other hand, EVONRL has a larger initialization overhead, but after that it can easily accommodate new edges. This is because every update is only related to the number of random walks affected and not the size of the network. This is an important observation, as it means that the benefit of EVONRL will be more stressed in larger networks.

6.6 Q6 Decision-Making Performance of EVONRL

In this experiment, we compare the two different strategies for deciding when to obtain a network representation, PERIODIC and ADAPTIVE. The experiment is performed using the *BlogCatalog* network and the changes in the network are related to edge addition. For presentation purposes, we limit the experiment to 1000 edges. The evaluation of this experiment is based on the number of random walk changes $RW^{t}_{t_{old}}$ between a random walk set obtained at time t (one edge is added at each time) and a previously obtained network representation as defined by each strategy. Results are shown in Figure 6.13. The PERIODIC strategy represents a "blind" strategy where new embeddings are obtained periodically (every 50 times steps or every 100 time steps). On the other hand, the ADAPTIVE

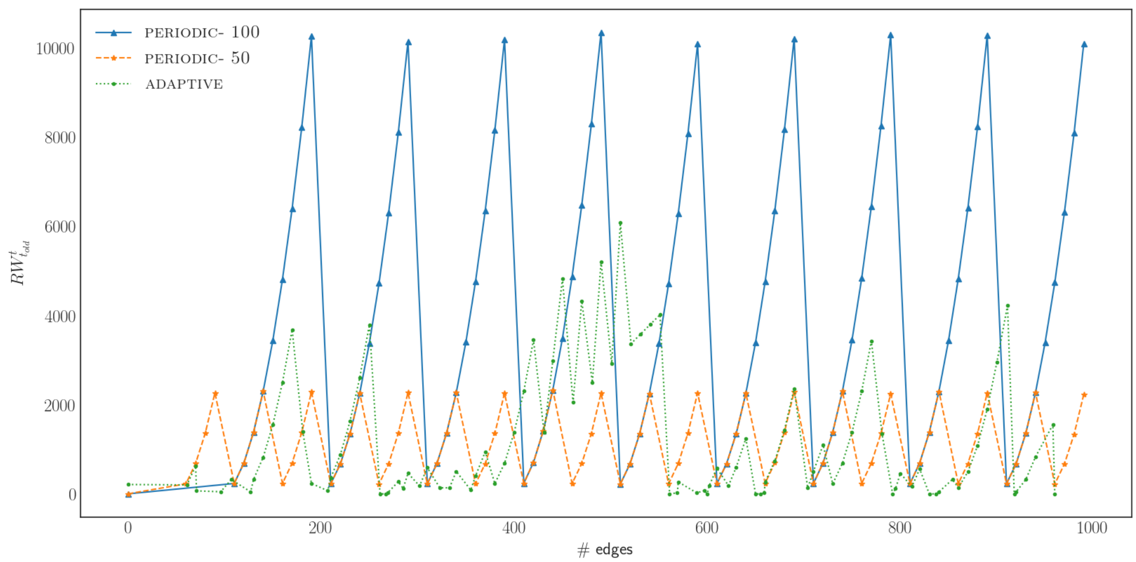


Figure 6.13: Comparative analysis of different strategies for determining when to obtain a network representation. The PERIODIC methods will obtain a new representation every 50 or 100 time steps (i.e., network changes). Our proposed method, ADAPTIVE, is combining a *peak detection* method and a *cumulative changes* cut-off method to determine the time to obtain a new network representation. As a result it is able to make more informed decisions and perform better. This is depicted by smaller (on average) changes of the $RW^t_{t_{old}}$, which implies that a more accurate network representation is available for down-stream network mining tasks.

method is able to make informed decisions as it monitors the importance of every edge added in the network. The ADAPTIVE method is basing its decisions on the a peak detection method ($\tau = 3.5$) and a method that monitors cumulative effects due to a number of changes ($cutoff = 4000$). As a result, ADAPTIVE is able to perform much better, as depicted by many very low values in the $RW^t_{t_{old}}$.

7 Conclusions

Our focus in this work is on learning representations of evolving networks. To extend static random walk based network representation methods to evolving networks, we proposed a general framework for updating random walks as new edges are arriving in the network. The updated random walks leverage time and space efficiency of inverted indexing methods. By indexing an initial random walk in the network and keep updating it based on the upcoming changes, we manage to always keep a valid set of random walks with minimum possible divergence from the initial random walk set. Our proposed method, EVONRL, utilizes the continuously valid set of random walks to obtain new network representations that respect the changes that occurred in the network. We demonstrated that our proposed method, EVONRL is both accurate and fast. We also discussed the interesting trade-off between time performance and accuracy when obtaining concurrent network representations. Determining the right time for obtaining a network embedding is a challenging problem. We demonstrated that simple strategies for monitoring the changes that occur in the network can provide support in decision making. Overall, the methods presented are easy to understand and simple to implement. They can also be easily adopted in diverse domains and applications of network mining.

7.1 Limitations

EvoNRL inherits the short-comings of static random walk-based network representation learning methods. Although random walk-based methods have a superior performance in a number of settings, the skip-gram direct encoding leads to a number of drawbacks [16]:

- There is no parameter sharing in skip-gram, which means the number of parameters necessarily grows as $\mathcal{O}(|V|)$.
- They fail to leverage node attributes during the embedding. Nodes can have attributes in large graphs and it can be informative of the node's position and role in the graph.

7.2 Future Work

We proposed a general framework to update random walks in an evolving network. We aim to continue our work in the following directions:

- EvoNRL uses random walks as a sampling strategy of the network. Random walks represent each snapshot of the network regardless of time. One aspect of integration of time into EvoNRL is generalizing time-invariant random walks to temporal random walks [43], which are able to traverse links over time. Moreover, there are many studies on random walks on time-series such as [38]. These 1-dimensional random walks can be used as a sampling strategy of a time-series of arriving edges where two consecutive traverses may not necessarily refer to primary links of the network.

- We aim to slightly modify the skip-gram objective function to leverage the frequency of the network's events and integrate the time dimension of the network into our representation learning methodology.

7.3 Reproducibility

We make source code and data sets used in the experiments publicly available[5] to encourage reproducibility of results.

[5] https://github.com/farzana0/EvoNRL

Bibliography

[1] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed large-scale natural graph factorization. ACM, 2013.

[2] N. K. Ahmed, R. Rossi, J. B. Lee, X. Kong, T. L. Willke, R. Zhou, and H. Eldardiry. Learning role-based graph embeddings. *arXiv preprint arXiv:1802.02896*, 2018.

[3] M. Antoniak and D. Mimno. Evaluating the stability of embedding-based word similarities. *TACL*, 6:107–119, 2018.

[4] V. Barnett and T. Lewis. *Outliers in statistical data; 3rd ed.* Wiley series in probability and mathematical statistics. Wiley, Chichester, 1974.

[5] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE TPAMI*, 35(8):1798–1828, 2013.

[6] B.-J. Breitkreutz, C. Stark, T. Reguly, L. Boucher, A. Breitkreutz, M. Livstone, R. Oughtred, D. H. Lackner, J. Bähler, V. Wood, et al. The biogrid interaction database: 2008 update. *Nucleic acids research*, 36(suppl_1):D637–D640, 2007.

[7] H. Cai, V. W. Zheng, and K. Chang. A comprehensive survey of graph embedding: Problems, techniques and applications. *IEEE TKDE*, 2018.

[8] S. Cao, W. Lu, and Q. Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 891–900. ACM, 2015.

[9] B. P. Chamberlain, J. Clough, and M. P. Deisenroth. Neural embeddings of graphs in hyperbolic space. *arXiv preprint arXiv:1705.10359*, 2017.

[10] H. Chen, B. Perozzi, Y. Hu, and S. Skiena. Harp: Hierarchical representation learning for networks. *arXiv preprint arXiv:1706.07845*, 2017.

[11] N. Craswell and M. Szummer. Random walks on the click graph. In *SIGIR*, pages 239–246. ACM, 2007.

[12] L. Du, Y. Wang, G. Song, Z. Lu, and J. Wang. Dynamic network embedding: an extended approach for skip-gram based network embedding. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2086–2092. AAAI Press, 2018.

[13] P. Goyal, N. Kamra, X. He, and Y. Liu. Dyngem: Deep embedding method for dynamic graphs. *IJCAI Workshop on Representation Learning for Graphs*, 2018.

[14] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *KDD*, pages 855–864, 2016.

[15] W. L. Hamilton, J. Leskovec, and D. Jurafsky. Diachronic word embeddings reveal statistical laws of semantic change. *CoRR*, abs/1605.09096, 2016.

[16] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bulletin*, 2017.

[17] F. Heidari and M. Papagelis. Evonrl: Evolving network representation learning based on random walks. In *International Workshop on Complex Networks and their Applications*, pages 457–469. Springer, 2018.

[18] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] H. Hotelling. Relations between two sets of variates. In *Breakthroughs in statistics*, pages 162–190. Springer, 1992.

[20] M. Jiang, A. W.-C. Fu, and R. C.-W. Wong. Reads: A random walk approach for efficient and accurate dynamic simrank. *Proc. VLDB Endow.*, 10(9):937–948, 2017.

[21] V. M. Kenkre, E. W. Montroll, and M. F. Shlesinger. Generalized master equations for continuous-time random walks. *Journal of Statistical Physics*, 9:45–50, Sept. 1973.

[22] Y. Kim, Y. Chiu, K. Hanaki, D. Hegde, and S. Petrov. Temporal analysis of language through neural language models. *CoRR*, abs/1405.3515, 2014.

[23] S. Kornblith, M. Norouzi, H. Lee, and G. Hinton. Similarity of neural network representations revisited. *ICML*, 2019.

[24] Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series, the handbook of brain theory and neural networks, 1998.

[25] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[26] J. Li, H. Dani, X. Hu, J. Tang, Y. Chang, and H. Liu. Attributed network embedding for learning in a dynamic environment. In *CIKM*, pages 387–396, 2017.

[27] L. Lovász. Random walks on graphs. *Combinatorics, Paul erdos is eighty*, 2(1-46):4, 1993.

[28] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. 2013.

[29] 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.

[30] A. Morcos, M. Raghu, and S. Bengio. Insights on representational similarity in neural networks with canonical correlation. In *Advances in Neural Information Processing Systems*, pages 5727–5736, 2018.

[31] M. E. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, 2003.

[32] G. H. Nguyen, J. B. Lee, R. A. Rossi, N. K. Ahmed, E. Koh, and S. Kim. Continuous-time dynamic network embeddings. In *Companion of the The Web Conference 2018 on The Web Conference 2018*, pages 969–976. International World Wide Web Conferences Steering Committee, 2018.

[33] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114. ACM, 2016.

[34] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[35] W. Paul and J. Baschnagel. *Stochastic Processes: From Physics to Finance*. Springer Berlin Heidelberg, 2010.

[36] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. *CoRR*, abs/1403.6652, 2014.

[37] B. Perozzi, V. Kulkarni, and S. Skiena. Walklets: Multiscale graph embeddings for interpretable network classification. 2016.

[38] P. Perron. Trends and random walks in macroeconomic time series: Further evidence from a new approach. *Journal of economic dynamics and control*, 12(2-3):297–332, 1988.

[39] M. Raghu, J. Gilmer, J. Yosinski, and J. Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6076–6085. Curran Associates, Inc., 2017.

[40] J. Ramsay, J. ten Berge, and G. Styan. Matrix correlation. *Psychometrika*, 49(3):403–423, 1984.

[41] Z. Reza and L. Huan. Social computing data repository.

[42] M. Rudolph and D. Blei. Dynamic embeddings for language evolution. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web*, pages 1003–1011. International World Wide Web Conferences Steering Committee, 2018.

[43] M. Starnini, A. Baronchelli, A. Barrat, and R. Pastor-Satorras. Random walks on temporal networks. *Physical Review E*, 85(5):056115, 2012.

[44] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. LINE: large-scale information network embedding. *CoRR*, abs/1503.03578, 2015.

[45] R. Trivedi, H. Dai, Y. Wang, and L. Song. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *ICML*, volume 70, pages 3462–3471, 2017.

[46] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1225–1234, New York, NY, USA, 2016. ACM.

[47] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[48] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang. Network representation learning with rich text information. In *IJCAI*, pages 2111–2117, 2015.

[49] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *KAIS*, 42(1):181–213, 2015.

[50] W. Yu, W. Cheng, C. C. Aggarwal, K. Zhang, H. Chen, and W. Wang. Netwalk: A flexible deep embedding approach for anomaly detection in dynamic networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2672–2681. ACM, 2018.

[51] D. Zhang, J. Yin, X. Zhu, and C. Zhang. Network representation learning: A survey. *IEEE Transac. on Big Data*, 2018.

[52] Y. Zuo, G. Liu, H. Lin, J. Guo, X. Hu, and J. Wu. Embedding temporal network via neighborhood formation. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2857–2866. ACM, 2018.