EvoNRL: Evolving Network Representation Learning based on Random Walks

Farzaneh Heidari and Manos Papagelis

York University, Toronto ON M3J1P3, Canada
{farzanah, papaggel}@eecs.yorku.ca

Abstract. Large-scale network mining and analysis is key to revealing the underlying dynamics of networks. Lately, there has been a fast-growing interest in learning random walk-based low-dimensional continuous representations of networks. While these methods perform well, they can only operate on static networks. In this paper, we propose a random-walk based method for learning representations of evolving networks. 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. This way we are able to continuously learn a new mapping function from the new network to the existing low-dimension network representation. A thorough experimental evaluation is performed that demonstrates that our method is both accurate and fast, for a varying range of conditions.

Keywords: network representation, evolving networks, random walks

1 Introduction

Large-scale network mining and analysis is key to revealing the underlying dynamics of networks, not easily observable before. Traditional approaches to network mining and analysis inherit a number of limitations; typically algorithms do not scale well (due to ineffective representation of network data) and require domain-expertise. More recently and to address the aforementioned limitations, there has been a growing interest in learning low-dimensional representations of networks, also known as network embeddings. A comprehensive coverage can be found in the following surveys [5, 9, 19]. 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 [13], 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 [15] and node2vec [7]. In this work, we collectively refer to these random-walk based methods as StaticNRL. A typical StaticNRL method, is operating in two steps:

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

(ii) *walks* are provided as input to an optimization problem that is solved using variants of Stochastic Gradient Descent using a deep neural network [3]. The outcome is a set of d-dimensional representations, one for each node.

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.

The main objective of this paper 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 (lack of stability), as well as, the non-deterministic nature of the learning process (Section 2). Therefore the naive approach would be inadequate for learning representations of evolving networks.

In contrast to the naive approach, we propose a novel algorithm, EvoNRL, for Evolving Network Representation Learning based on random walks. 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 (Section 3). As a result, we are able to design a strategy for continuously learning 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 the network embedding (Section 4). A thorough experimental evaluation on synthetic and real data sets demonstrates that the proposed method offers substantial time performance gains without loss of accuracy (Section 5). In addition, the method is generic, so it can accommodate a variant of random-walk based methods and the needs of different domains and applications.

2 Instability of StaticNRL Methods

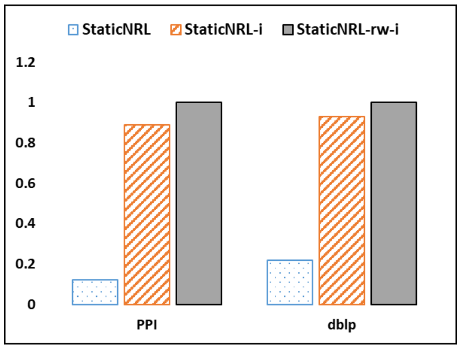
In this paragraph, we present a systematic evaluation of the stability of the StaticNRL methods, similar to the one presented in [2]. The evaluation aims to motivate our approach to address the problem of interest. Intuitively, a stable embedding method is one in which successive runs of it on the same network would learn the same (or similar) embedding. StaticNRL methods are to a great degree dependent on two random processes: (i) the set of random walks collected, and (ii) the random 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:

– Embedding Similarity: Cosine similarity is a popular similarity measure for real-valued vector space models [8, 11]. Formally, given the vector representations $\mathbf{n_i}$ and $\mathbf{n_i'}$ of the same node n_i in two network embeddings obtained at two different times, their cosine similarity is: $sim(\mathbf{n_i}, \mathbf{n_i'}) = cos(\theta) = \frac{\mathbf{n_i} \cdot \mathbf{n_i'}}{\|\mathbf{n_i}\| \|\mathbf{n_i'}\|}$. We can extend the similarity to two network embeddings $\mathscr{E}$ and $\mathscr{E}'$ by summing and normalizing over all nodes: $sim(\mathscr{E}, \mathscr{E}') = \sum_{i \in V} sim(\mathbf{n_i}, \mathbf{n_i'})/|V|$.
– Embedding Distance: Given a graph $G = (V, E)$, a network embedding is a mapping $f : V \to \mathbb{R}^d$, where $d \ll |V|$. Let $F_t(V) \in \mathbb{R}^{|V| \times d}$ be the matrix of all node representations at time t. Then, similarly to the approach in [6], the distance of two embeddings $\mathscr{E}$, $\mathscr{E}'$ is: $distance(\mathscr{E}, \mathscr{E}') = ||F(V) - F'(V)||_F$.

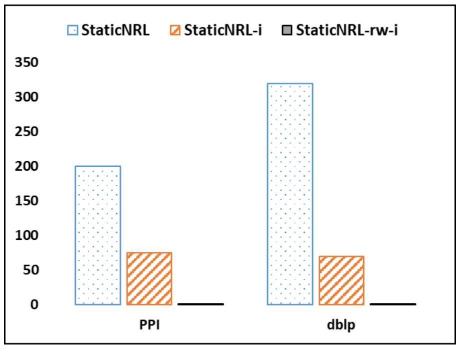
Experimental Scenario: We design a controlled experiment on two real-world networks, namely Protein-Protein-Interaction (PPI) [4] and a collaboration network (dblp) [18] that aims to evaluate the effect of the two random processes (set of random walks, initialization weights) in the network embeddings. In these experiments, we compare three settings. For each setting, we run StaticNRL on a network (using parameter values: $r = 10$, $l = 10$, $k = 5$) two times (while there have been no change in the network), and compute the cosine similarity and the matrix distance of the two embeddings $\mathscr{E}$, $\mathscr{E}'$. We repeat the experiment 10 times and report averages. The three settings are:

– StaticNRL: independent set of random walks; random initialization weights.
– StaticNRL-i: independent set of random walks; same initialization weights.
– StaticNRL-rw-i: same set of random walks; same initialization weights.

Results & Implications: The results of the experiment are shown in Fig. 1a (cosine similarity) and Fig. 1b (matrix distance). They show that when we employ the same set of random walks and the same initialization in consecutive runs of StaticNRL, we are able to obtain the same embedding (as depicted by the $sim(\cdot, \cdot) = 1$ in Fig 1a or $distance(\cdot, \cdot) = 0$ in Fig. 1b). However, when random walks and/or random initialization are employed in consecutive runs of a StaticNRL method, then the embedding can be shifted (lack of stability) despite the fact that there is no actual change in the network topology. Most of similar work in the literature correct this problem by applying an alignment method [8] that aims to rotationally align two embeddings. 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 both random processes. Therefore, changes that occur in the evolving network topology will not be easily interpretable by the changes observed in the network embedding (since differences might incorporate changes due to the two random processes). These observations highlight the challenges of employing StaticNRL methods for learning representations of evolving networks, which is the focus of this work.



(a) embedding similarity



(b) embedding distance

Fig. 1. Instability of the StaticNRL methods when applied on the same network.

3 Dynamic Random Walks Framework

In Section 2, we established the instability of random-walk based methods when they are repeatedly applied to the same static network. This motivated our key idea to address the problem: maintain a set of random walks that are consistently valid with respect to the network topology changes. If we are able to do so, we can effectively eliminate the effect of the random processes by, first, preserving, as much as possible, the original random walks that haven't been affected by the network changes. Then, by initializing the model with a previous run's initialization [11]. The main advantage of this approach is that any changes observed in the embedding of an evolving network will be more interpretable. Stemming from our key idea, in this Section we describe a general framework and a novel method that allows to incrementally update a set of random walks obtained on a network $G_t(V_t, E_t)$ at time t so that they remain valid at time $t' > t$ and network $G_{t'}(V_{t'}, E_{t'})$. The most common change in an evolving network consists of either adding a new edge or deleting an existing edge. While our method is able to handle both cases (with minor variations), for the rest of the manuscript we discuss only the case of edge addition, due to space limitations. Therefore, $G_{t'}$ represents an augmentation of G_t, where $V_{t'} = V_t$ and $E_{t'} = E_t \cup E^+$, and E^+ represents the set of the new edges in G_t.

Dynamic Updates 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. By default, this method is configured to perform $r = 10$ random walks per node, each of length $l = 80$. 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). Now, assume that a single new edge $e_{ij} = (node_i, node_j)$ arrives in the network at time $t+1$, so

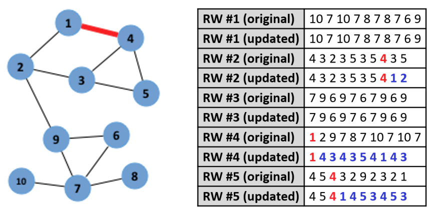
[1]node2vec; code is available at https://github.com/aditya-grover/node2vec

$E_{t+1} = E_t \cup (node_i, node_j)$. 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}.

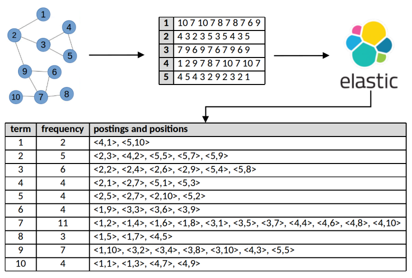
Operation 1: We want to update the set RW_t to contain the new edge $(node_i, node_j)$. We explain the update process for $node_i$; the same process is followed for $node_j$. First, we need to find all the random walks $walks_i \in RW_t$ that include $node_i$. Then, we need to update them so as to reflect 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. 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. A faster approach 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 employ the faster approach for the updates. 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 of a node in random walks.

Operation 2: Once a new edge $(node_i, node_j)$ is contained in an existing random walk, it renders the rest of it obsolete. We 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 $node_j$ with length $l_{sim} = l - (Ind_i + 1)$, where $0 \leq Ind_i \leq l - 1$ is the index of $node_i$ in the currently updated random walk. Fig. 3 presents an illustrative example of how updates of random walks work. 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



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 adhere to the updates of the network.



1	10 7 10 7 8 7 8 7 6 9
2	4 3 2 3 5 3 5 4 3 5
3	7 9 6 9 7 6 7 9 6 9
4	1 2 9 7 8 7 10 7 10 7
5	4 5 4 3 2 9 2 3 2 1

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>

(b) Example inverted random walk index. Given a graph, five random walks are performed and indexed.

Fig. 2. Illustrative examples of the update method in both naive and faster approach.

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 of the 4th random walk 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 on the 2nd and 5th random walk, 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 set of the random walks RW_t. In addition, lines 11 and 19 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. In the next paragraph, we describe how these two slow operators, *Query* and *UpdateRandomWalks*, 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. The indexing technologies we will employ scale well to very large number of random walks.

Efficient Query & Update of Random Walks: Updating random walks methods presented in the previous paragraph are accurate. However, they depend on operators *Query* and *UpdateRandomWalks* that are computationally expensive and are not scalable. The most expensive operation is to search and update the random walks RW_t with the occurrences of $node_i$ and $node_j$ of the new edge $(node_i, node_j)$. To address these shortcomings, our framework relies on popular open-source indexing and searching technologies. 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. 3 provides an example of a small inverted random walk index. We also describe how to create the index and use it in our setting.

Algorithm 1 Update RW

```
1:  procedure UpdateWalks
2:      $walks_i \leftarrow$ Query($node_i$)
3:      $p_i \leftarrow \frac{1}{d^i}$
4:      $p_j \leftarrow \frac{1}{d^j}$
5:      $s_i \leftarrow$ Sample($walks_i$, $p_i$)
6:      if $len(s_i) > 0$ then
7:          for wk in $s_i$ do
8:              $Ind_i \leftarrow$ Position($node_i$, $wk$)
9:              $l_{sim} = l - (Ind_i + 1)$
10:             $wk[Ind_i$+1:$] \leftarrow$ SimulateWalk($node_j$, $l$)
11:     UpdateRandomWalks()
12:     $walks_j \leftarrow$ Query($node_j$)
13:     $s_j \leftarrow$ Sample($walks_j$, $p_j$)
14:     if $len(s_j) > 0$ then
15:         for wk in $s_j$ do
16:             $Ind_j \leftarrow$ Position($node_j$, $wk$)
17:             $l_{sim} = l - (Ind_j + 1)$
18:             $wk[Ind_j$+1:$] \leftarrow$ SimulateWalk($node_i$, $l$)
19:     UpdateRandomWalks()
```

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 $(x \rightarrow y \rightarrow z)$ 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. Besides searching and retrieving all random walks that contain a specific $node_i$, the index I^{RW} can provide useful quantities of interest like the frequency of $node_i$ in a set of random walks, and the position of $node_i$ in a random walk.

Updating Random Walks: We rely on the index I^{RW} for any *UpdateRandomWalks* operation. While querying an inverted index is a fast process, updating it is slower. Therefore, the performance of our methods is dominated by the number of random walk updates required. Still, our methods would perform multitude of times faster than StaticNRL methods. A detailed analysis of this issue is provided in Section 5. Additional optimizations are available as a result of employing an inverted index. For instance, we can take advantage of bulk updates. This will sacrifice accuracy for speed that might be preferable in some applications.

4 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 the evolving network. The process begins by obtaining an initial network embedding. For that, we resort to the StaticNRL method of choice. EvoNRL has the overhead of first indexing the set of initial random walks RW. At this time only ($t = 0$), we initialize the skip-gram model, but we store and re-use the same initialization weights for the needs of subsequent embeddings. As new edges are arriving, EvoNRL performs the necessary updates as described earlier. At each time t a valid set of random walks RW_t is available that can be used to obtain an updated network embedding. While re-embedding the network at every time t will result in more accurate embeddings, we only need to do this once in a while. The timing of the embedding is domain-specific and relates to the trade-off between accuracy and performance that an application can accommodate. In Section 5 we present experiments that can support the decision making process. For completeness, we describe next how to obtain an embedding of the evolving network at time t, given a set of random walks RW_t.

Learning Embeddings given $\mathbf{RW_t}$: Given a 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 each row is the vector representation of a node and d is the vector's dimension. The context of each node n_i is found using the valid RW_t set, similar to works [7, 15]. To obtain an embedding we optimize the skip-gram with negative sampling objective, using stochastic gradient decent, so that: $Pr(n_j|\mathbf{n_i}) \propto \exp(\mathbf{n_j^T n_i})$, where $\mathbf{n_i}$ 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.

5 Experimental Evaluation

In this section, we experimentally evaluate the performance of our dynamic random walk framework and EvoNRL. We aim to answer the following questions:

– **Q1** How the importance of a new edge affects the random walks update time?
– **Q2** How the network topology affects the number of random walks updated?
– **Q3** What is the time performance of EvoNRL?
– **Q4** What is the accuracy performance of EvoNRL?

Environment: All experiments are conducted on a workstation with 8x Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz and 64GB memory. For the embeddings, we employ the gensim implementation of the skip-gram model[2]. We set the context-size $k = 5$ and the number of dimensions $d = 128$. We use Python 3.6.

Data: We experiment with real networks coming from a Protein-to-Protein interaction network (PPI [4]) and a social network of bloggers (BlogCatalog [16]). In addition, we experiment with a set of synthetic networks of different topologies, obtained using the Watts-Strogatz model [14]. In particular, we set the model's rewiring probability parameter p, so as to obtain a Lattice ($p = 0.0$), a Small-world ($p = 0.5$) and an Erdos-Reyni ($p = 1.0$) network, respectively. Details are shown in Table 1.

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

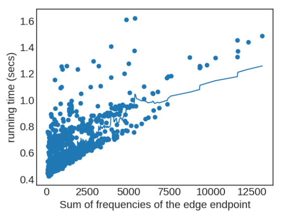
name	type	# nodes	# edges	description
PPI	real	$3,890$	$76,584$	50 different labels
BlogCatalog	real	$10,312$	$333,983$	39 different labels
{Lattice, Small-world, ER}	synthetic	$10,000$	$70,000$	$p = \{0, 0.5, 1.0\}$, respectively

Table 1. Description of the network data sets employed in the experimental evaluation.

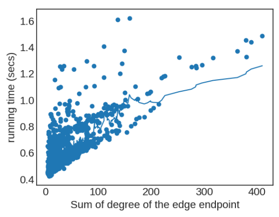
Q1 Effect of New Edge Importance: It is easy to see that even a single new edge can have a dramatic effect in the number of random walks affected. Apparently, that number controls the time needed to update the affected random walks in our framework. In this set of experiments we perform a systematic analysis of the effect of the importance of an arriving edge to the time required for the update. Importance of an incoming edge $e_{ij}^{t+1} = (n_i, n_j)$ at time $t+1$ in a network can be defined in different ways. Here, we define three metrics of edge importance, based on properties of the nodes n_i and n_j that form the endpoints of an arriving edge. These include the sum of frequencies of edge endpoints in RW_t, sum of the node degrees of edge endpoints in G_t and sum of the node-betweenness of edge endpoints in G_t. Results are presented in Fig. 3. The first observation is that important incoming edges are more expensive to update, 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. 3a, 3b, 3c), so fast updates are more common. Finally, the behavior of sum of frequencies (Fig. 3a) and sum of degrees (Fig. 3b) 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 related to shortest paths and not just paths (which are traversed in randw3om walks).

Q2 Effect of Network Topology: We evaluate the effect of randomly adding new edges in networks of different topologies, but same size ($|V| = 10,000$). For each case, we report the number of the random walks that need to be updated. Fig. 4 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; these networks have small diameter, therefore every node is easily accessible from any other node. In contrast, Lattices have larger diameter and nodes are more equally distributed in random walks.

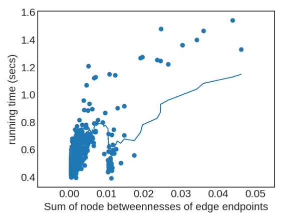
Q3 Time Performance of EvoNRL: To evaluate the time performance of EvoNRL we run experiments on two Small-world networks (Watts-Strogatz ($p = 0.5$)) of different size ($|V| = \{1000, 10000\}$). We evaluate EvoNRL against a standard StaticNRL method from the literature [7]. Both algorithms start with the same set of random walks RW. As new edges are arriving, StaticNRL needs to learn a new network representation by re-simulating 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 it just needs to perform the necessary updates as described earlier. Fig. 5 shows the results, where it can be seen that the performance of StaticNRL is linear to the number of new edges. At the same



(a) sum of frequencies



(b) sum of node degrees



(c) sum of node-betweenness

Fig. 3. Effect of the new edge importance to the running time of updates (PPI is used).

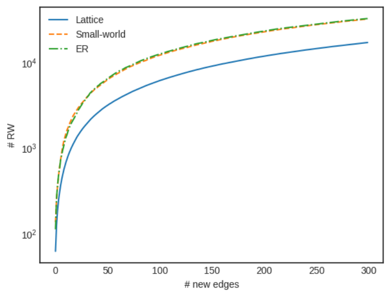


Fig. 4. Effect of network topology.

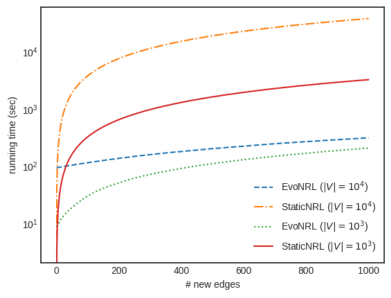
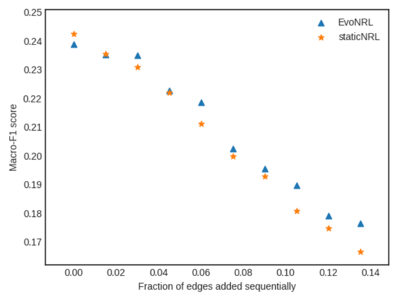


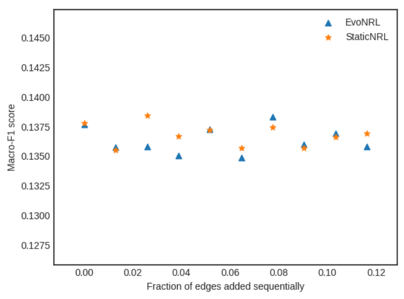
Fig. 5. Time performance of EvoNRL.

time, EvoNRL can 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 it needs to simulate a 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.

Q4 Accuracy Performance of EvoNRL: We run experiments that demonstrate that EvoNRL has a similar accuracy to that obtained by StaticNRL, when it is run again and again on instances of an evolving network. The experiment is designed as follows: we randomly add new edges to the original network (BlogCatalog, PPI) and evaluate the accuracy on a downstream data mining task: multi-label classification. In our multi-label classification 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. Adding new edges will impact the network embedding and thus the overall accuracy of the classification results. It is important to note here that we only care about observing similar trends in the accuracy results of both methods, and not about the actual



(a) BlogCatalog



(b) PPI

Fig. 6. Accuracy performance of EvoNRL for the BlogCatalog and the PPI network.

accuracy values. For both experiments we report the $Macro - F_1$ accuracy of the multi-label classification task as a function of the fraction of new edges added. For StaticNRL, since it is sensitive to the new set of random walks, we run it 10 times and report the averages. Fig. 6 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. 6a) and the PPI (Fig. 6b) networks. It can be seen that the accuracy of the two methods remains similar as more edges are added. This provides strong evidence that our random walk updates are accurate and able to reproduce the accuracy results obtained by applying a StaticNRL method on multiple instance of the evolving network.

6 Related Work

A comprehensive coverage of methods for learning network representations of static networks can be found in [5,9,19]. Work on learning representations of dynamic networks often apply static methods to snapshots of an evolving network [8]. Similarly, graph factorization approaches attempt to learn the embedding of dynamic graphs by smoothing over consecutive snapshots [1]. DANE [12] is a dynamic representation framework that focuses on attributed networks. Know-Evolve [17] proposes an entity embedding method of an evolving knowledge-graph based on multivariate event detection. EvoNRL does not need to operate on snapshots of the evolving network; instead, it directly learns the evolving network representations by monitoring the changes in the topology. Our work is also related to work on dynamic random walks. For instance, READS [10] is an indexing scheme for Simrank computation in dynamic graphs. EvoNRL has different semantics, sampling strategy and application focus to READS.

7 Conclusions

Our focus in this paper 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. The updated random walks leverage time and space efficiency of inverted indexing methods. 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. Therefore, it can be successfully employed in a number of predictive tasks that arise in the study of networks that evolve over time.

References

1. Ahmed, A., Shervashidze, N., Narayanamurthy, S., Josifovski, V., Smola, A.J.: Distributed large-scale natural graph factorization. ACM (2013)
2. Antoniak, M., Mimno, D.: Evaluating the stability of embedding-based word similarities. TACL 6, 107–119 (2018)
3. Bengio, Y., Courville, A., Vincent, P.: Representation learning: A review and new perspectives. IEEE TPAMI 35(8), 1798–1828 (2013)
4. Breitkreutz, B.J., Stark, C., Reguly, T., Boucher, L., Breitkreutz, A., Livstone, M., Oughtred, R., Lackner, D.H., Bähler, J., Wood, V., et al.: The biogrid interaction database: 2008 update. Nucleic acids research 36(suppl_1), D637–D640 (2007)
5. Cai, H., Zheng, V.W., Chang, K.: A comprehensive survey of graph embedding: Problems, techniques and applications. IEEE TKDE (2018)
6. Goyal, P., Kamra, N., He, X., Liu, Y.: Dyngem: Deep embedding method for dynamic graphs. IJCAI Workshop on Representation Learning for Graphs (2018)
7. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: KDD, pp. 855–864 (2016)
8. Hamilton, W.L., Leskovec, J., Jurafsky, D.: Diachronic word embeddings reveal statistical laws of semantic change. CoRR abs/1605.09096 (2016)
9. Hamilton, W.L., Ying, R., Leskovec, J.: Representation learning on graphs: Methods and applications. IEEE Data Eng. Bulletin (2017)
10. Jiang, M., Fu, A.W.C., Wong, R.C.W.: Reads: A random walk approach for efficient and accurate dynamic simrank. Proc. VLDB Endow. 10(9), 937–948 (2017)
11. Kim, Y., Chiu, Y., Hanaki, K., Hegde, D., Petrov, S.: Temporal analysis of language through neural language models. CoRR abs/1405.3515 (2014)
12. Li, J., Dani, H., Hu, X., Tang, J., Chang, Y., Liu, H.: Attributed network embedding for learning in a dynamic environment. In: CIKM, pp. 387–396 (2017)
13. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: NIPS, pp. 3111–3119 (2013)
14. Newman, M.E.: The structure and function of complex networks. SIAM review 45(2), 167–256 (2003)
15. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: Online learning of social representations. CoRR abs/1403.6652 (2014)
16. Reza, Z., Huan, L.: Social computing data repository
17. Trivedi, R., Dai, H., Wang, Y., Song, L.: Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In: ICML, vol. 70, pp. 3462–3471 (2017)
18. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. KAIS 42(1), 181–213 (2015)
19. Zhang, D., Yin, J., Zhu, X., Zhang, C.: Network representation learning: A survey. IEEE Transac. on Big Data (2018)