1 Introduction

Information network is a common abstraction for many a dataset that emerges nowadays including social network, protein-to-protein network, scholar network, etc[2]. To analyze and leverage such a huge volume of data, there has been a comprehensive collection of algorithms such as Support Vector Machine (SVM), Deep Neural Network, K-Means and so on. However, there is a representation gap between the format of information network and input of most of machine learning algorithms. Based on classic graph theory, we usually use adjacent list, adjacent matrix or edge list to represent an information network and to compromise with the heterogeneity of some networks, which are call Heterogeneous Information network[3], auxiliary data structures may also be applied. On the other hand, algorithms mentioned earlier use real-valued vectors or matrices as their inputs. Therefore, the research community has defined the problem of converting an information network to a real-valued matrix as Network Embedding, or Network Representation Learning.

There are several classes of network embedding algorithms according to Palash and Emilio’s survey[4] including factorization based, random-walks based and deep learning based. Thanks to its scalability and trivially parallelizable nature, random-walks based algorithms have formed a trend since the earliest [9] paper which cleverly borrowed idea from word2vec[8], a natural language processing techniques. And most recent research is focusing on effectiveness, that is how to embed an information network to a matrix while preserving its structural information.

Nevertheless, there is lack of efficiency studies and an efficient yet unified framework for developers and researchers to leverage and investigate network embedding algorithms. To tackle these two problems, we propose Fast2Vec, an efficient network embedding framework optimized for cache access.

The remaining of the paper will first describe random-walk based network embedding algorithms, especially the relation between word2vec and random-walk, as well as the extended unified framework based on Palash and Emilio’s work. The second half of the paper will be dedicated to how do we re-organize the embedding algorithms so that we can make it more cache friendly and therefore improve efficiency.

2 Random-walk Based Network Embedding

In this section, we first define the network embedding problem and then illustrate its relation with word embedding and the noble word2vec algorithm. Therefore, we will be able to describe most of the current embedding algorithms under a unified framework. In this final year project, the implementation will also be based on the framework so future users can extend new algorithms in several lines of codes.

2.1 Problem Definition

Let $G = (V,E)$ be a given network and $f : V \rightarrow \mathbb{R}^d$ be the embedding function which maps a vertex in the network to a $d$ dimension real-valued vector. Therefore, it is essential for $f$ to preserve the structural information of $G$ and it is nothing but given $f(v)$ we will be able to recover its neighborhood $N(v)$. The formal objective function is as below
\[
\max \sum_{v \in V} \log \text{Pr}(N(v) | f(v))
\] (1)

It implies we want to maximize probability of occurrence of neighbor of \(v\) conditioned on its vector representation.

### 2.2 Word2Vec and Network Embedding

Word2Vec is a noble natural language processing algorithm to find vector representation of word in text data[8]. And DeepWalk is the first algorithm leveraging it to the network embedding problem. The rationale behind is the power law[9]. They found that distribution of vertices in a network follows power-law which much alike the distribution of words in natural languages and they have also plotted both YouTube social network and Wikipedia Article Text to empirically show their findings so we can apply natural language processing algorithms in the network embedding problem. As skip-gram, one variance of word2vec, is effective to train a model that maximize probability of occurrence of words related to the input word \(w\) conditioned on its vector representation, DeepWalk modified it to work with network data.

#### 2.2.1 Skip-Gram Model

Before showing how can we utilize skip-gram into a network, let’s take a closer look at it first.

To evaluate \(\text{Pr}(N(v) | f(v))\) we use conditional probability and assume all the neighbors are distributed independently so Eq-1 can be converted to

\[
\max \sum_{v \in V} \log \text{Pr}(N(v) | f(v)) = \max \sum_{v \in V} \log \text{Pr}(u_1 \ldots | v) = \max \sum_{v \in V} \log \left( \prod_{u \in N(v)} \text{Pr}(u | v) \right) = \max \sum_{v \in V} \sum_{u \in N(v)} \log \text{Pr}(u | v)
\] (2)

Since in network embedding, we are doing nothing but map an object into a vector, we can denote the embedding function as

\[f(u) = W_u\]

where \(W\) is a \(|V| \times K\) matrix, \(K\) is the feature dimension, and \(W_u\) is \(u\)'s vector representation. Because word2vec also requires a hidden layer, we will have \(f'\) and \(W'\) to represent them accordingly.

With vector representation of object \(u\) and \(v\), word2vec uses softmax function, a function takes a unified vector and normalized it to a probability distribution, to calculate conditional probability which yields

\[\text{Pr}(u | v) = \frac{\exp(W'_u \cdot W_v)}{\sum_j \exp(W'_j \cdot W_v)}\] (3)

However, it is expensive to evaluate the denominator of softmax function so Mikolov etal. [8] proposed negative sampling to simulate \(\log \text{Pr}(u | v)\)

\[\log \text{Pr}(u | v) \sim \log \sigma(W'_u \cdot W_v) + \sum_k \log \sigma(-W'_k \cdot W_v)\] (4)

where \(W'_k\) are vector representations of randomly selected negative examples. Using stochastic gradient descent on each source object \(v\) and context object \(u\), we can further derive the gradient based on Eq 4.
\[
\frac{\partial}{\partial W_u} = \frac{1}{\sigma(W_u \cdot W_v)} \cdot \frac{\partial \sigma(W_u \cdot W_v)}{\partial W_u} \\
= \frac{1}{\sigma(W_u \cdot W_v)} \cdot \sigma(W_u \cdot W_v) \cdot (1 - \sigma(W_u \cdot W_v)) \cdot \frac{\partial W_u \cdot W_v}{W_u} \\
= (1 - \sigma(W_u \cdot W_v)) \cdot W_v 
\]

\[
\frac{\partial}{\partial W_k} = \frac{1}{\sigma(-W_k \cdot W_v)} \cdot \frac{\partial \sigma(-W_k \cdot W_v)}{\partial W_k} \\
= \frac{1}{\sigma(-W_k \cdot W_v)} \cdot \sigma(-W_k \cdot W_v) \cdot (1 - \sigma(-W_k \cdot W_v)) \cdot \frac{\partial -W_k \cdot W_v}{W_k} \\
= (1 - \sigma(-W_k \cdot W_v)) \cdot (-W_v) \\
= -\sigma(W_k \cdot W_v) \cdot W_v 
\]

\[
\frac{\partial}{\partial W_v} = (1 - \sigma(W_u \cdot W_v)) \cdot W_u - \sum_k \sigma(W_k \cdot W_v) \cdot W_k 
\]

With the equation, we will be able to derive the update function for both \(W\) and \(W'\).

\[
W_v = W_v + \alpha \cdot \frac{\partial}{\partial W_v} \\
= W_v + \alpha \cdot \{(1 - \sigma(W_u \cdot W_v)) \cdot W_u - \sum_k \sigma(W_k \cdot W_v) \cdot W_k\} 
\]

\[
W_u = W_u + \alpha \cdot \frac{\partial}{\partial W_u} \\
= W_u + \alpha \cdot \{(1 - \sigma(W_u \cdot W_v)) \cdot W_v\} 
\]

\[
W_k = W_k + \alpha \cdot \frac{\partial}{\partial W_k} \\
= W_k + \alpha \cdot \{-\sigma(W_k \cdot W_v) \cdot W_v\} 
\]

With the update function above, the skip-gram model can be conclude as below

```plaintext
Algorithm 1 Skip-gram

function SkipGram(Text)
    W ← RandomInit()
    W' ← 0
    for sentence ← Text do
        for context ← sentence do
            for source ← Window(context, sentence) do
                W_v ← W_source
                W'_u ← W'_context
                \{W'_k\} ← NegativeSampling(Text)
                W, W' ← update(W_v, W'_u, \{W'_k\})
            end for
        end for
    end for
    return W, W'
end function
```
where \{W'_k\} is nothing but vector representations of \(k\) randomly picked negative examples. How to draw negative examples has been discussed in detail in \([8, 6]\) and will be omitted here as we will do it differently on network.

### 2.2.2 Relation between Text and Network

As described before, *skip-gram* model has been proven to be effective to train a model such that given an input vector, we could tell its neighbors. However, it is designed to work with text data where words are ordered while in a network we don’t naturally have such a sequence of objects. As studied in *DeepWalk* \([9]\), both words in text and vertices in network follows power distribution and this supports us to build a link between them. In Perozzi et al.’s \([9]\) work, they proposed to use random-walk as a substitute of sentences in text. For example, a sentence in *word2vec* may looks like

\[
\text{Sentence} = \{\text{I, eat, chocolate, everyday}\}
\]

While a random-walk in a scholarship network can be

\[
\text{RandomWalk} = \{\text{Jiawei Han, SIMOID, Pei Jian}\}
\]

And they both are ordered sequence so that we can train them in a similar fashion. In summary, to obtain an embedding of a network using *word2vec*, we can use the correspondence below

\[
\text{Sentence} \sim \text{RandomWalk},
\]

\[
\text{Text} = \{\text{Sentence}\} \sim \{\text{RandomWalk}\}
\]

Therefore the last fragment to complete the puzzle becomes how can we draw random-walks fitting our purpose.

### 2.2.3 Random-walks Generation

According to previous sections, we need to generate random-walks for each vertex in the network if we want to have vector representation for each of them. Therefore, we generate a fixed number of \(N\) random-walks starting from each vertex \(v\). To generate a random-walk, we need to define a conditional probability that given our current vertex \(v\) what is the probability of transferring to its neighbor \(u\) among all the neighbors of \(v\). Therefore we got

\[
\sum_{u \in \text{Neighbor}(v)} Pr(u|v) = 1 \tag{7}
\]

and it will ultimately define the random-walk strategy. Different conditional probability formulas are used in different random-walk based network embedding algorithm but they all follow the same procedure as described below.
**Algorithm 2** Random-walks Generation

```plaintext
function RANDOMWALKS Generation(N)
  RandomWalks ← ∅
  for vertex ← Vertices(N) do
    for i ← Range(N) do
      current ← vertex
      randomWalk ← ∅
      for l ← Range(L) do
        next ← Draw(current, N)
        randomWalk ← randomWalk ∪ {current}
        current ← next
      end for
      RandomWalks ← RandomWalks ∪ {randomWalk}
    end for
  end for
  return RandomWalks
end function
```

Note that Draw(current, N) choose a neighbor vertex of current vertex as the next vertex in the random-walk based on the conditional probability described before. In summary, we will generate N random-walks with length L for each vertex in the network.

With the RandomWalks we will be able to train our network with skip-gram model 1 with just a little modification.

**Algorithm 3** Network Skip-gram

```plaintext
function NETWORK SKIPGRAM(RandomWalks)
  W ← RandomInit()
  W′ ← 0
  for randomWalk ← RandomWalks do
    for context ← randomWalk do
      for source ← Window(context, randomWalk) do
        Wv ← Wsource
        W′ ← W′context
        {W′k} ← NegativeSampling(Text)
        W, W′ ← update(Wv, W′, {W′k})
      end for
    end for
  end for
  return W, W′
end function
```

As we can see, we just replace some variables in original skip-gram based on the relation described in section 2.2.2 to make it work with network data.

### 2.3 Unified Framework for Network Embedding

#### 2.3.1 Definition

With each component described in detail, we are now able to assemble all the pieces to form our framework of random-walk based network embedding.
Algorithm 4: Random-walk Based Network Embedding

```plaintext
def networkembedding(network):
    random_walks ← random_walks_generation(network)
    w, w' ← network_skip_gram(random_walks)
    return w, w'
end function
```

By having separated random-walks generation phase and skip-gram training phase, we will enjoy the following benefits.

1. **A unified interface to define random-walk strategy**. All established random-walk based algorithms like DeepWalk, Node2Vec and Metapath2Vec only differs in their random-walks generation phases which could be described with the formula in section 2.2.3. Therefore, we just need to implement the interface with few lines of codes to develop a new algorithm.

2. **Leverage works in skip-gram seamlessly**. The community has spent great effort on making word2vec effective and efficient. It includes lock free gradient descent[10], faster softmax computation[11, 5] and learning rate adjustment[7]. With such our framework, it requires less effort to be benefited from these technology.

And the following of the section will be dedicated to describe DeepWalk, Node2Vec and Metapath2Vec under the framework.

### 2.3.2 DeepWalk

It is the first paper introducing random-walk based network. It follows uniform distribution when doing random walks so each neighbor of the current vertex shares the same probability and the transition probability is

$$Pr_{\text{DeepWalk}}(u|v) = \frac{1}{|\text{Neighbor}(v)|}$$

Note that $v$ is the current vertex and $u$ is one of the neighbor of $v$, which makes it one of the candidates of the next vertex.

### 2.3.3 Node2Vec

It refines the DeepWalk model with a customizable transition probability to interpolate between BFS-like and DFS-like behavior.

$$\pi(u|v) = \frac{\alpha_{pq}(t,u)}{|\text{Neighbor}(v)|}$$

$$Pr_{\text{Node2Vec}}(u|v) = \frac{\pi(u|v)}{\sum_{x \in \text{Neighbor}(v)} \pi(x|v)}$$

Note that $t$ represents the vector we were before the current vertex $v$ and $\pi(u|v)$ is un-normalized probability biased by $\alpha_{pq}(t,u)$. $\alpha_{pq}$ is bias function based on the distance between $t$ and $u$ and $p,q$ are controlling parameters. It is defined as

$$\alpha_{pq}(t,u) = \begin{cases} 
  p, & d_{tu} = 0 \\
  1, & d_{tu} = 1 \\
  q, & d_{tu} = 2 
\end{cases}$$

Note that $d_{tu}$ denote the distance between $t$ and $u$. $p$ is called return parameter as a larger value of $p$ will make it the random-walk tend to return to where it comes while $q$ is called in-out parameter since a larger value of $q$ will encourage the random-walk to go further. In summary $p$ controls BFS-like behavior while $q$ controls DFS-like behavior.
2.3.4 Metapath2Vec

Unlike previous two strategies, Metapath2Vec is more selective when doing random-walk as it requires the result to be aligned with a user defined metapath and the rest is the same as the DeepWalk method.

\[
Pr_{\text{Metapath2Vec}}(u|v, P_1) = \frac{1}{|\text{Neighbor}_{P_1}(v)|}
\] (11)

Note that \( P = \{P_1, \ldots, P_n\} \) is a metapath, an ordered sequence of edge types. And \( \text{Neighbor}_{P_1}(v) \) is a subset of \( \text{Neighbor}(v) \) and constructed in the way described below

\[
\text{Neighbor}_{P_1}(v) = \{x | x \in \text{Neighbor}(v), \phi(v, x) = P_1\}
\] (12)

where \( \phi(v, x) \) is the type of the edge connecting \( v \) and \( x \).

3 Efficiency Study on Shared Memory

As proved empirically in the word2vec paper[8], the larger the training set is the more accurate the overall model will be so it is crucial to make the skip-gram phase efficient to work with larger data. This section will describe why and how we optimize our training algorithm.

3.1 Motivation

The skip-gram model with stochastic gradient descent[10] is trivially parallelizable but will suffer a lot from the cache miss and ping-pong update. In addition, the original version of network skip-gram3 uses only vector level operations which is just L1 level BLAS operation[1]. Therefore, it is very promising to increase cache-hit rate and leverage L1 BLAS operations to L3 operations which is matrix-wise. To analyze the problem, let’s first define following notations

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>Length of random-walk</td>
</tr>
<tr>
<td>N</td>
<td>Number of random-walks starting from each node</td>
</tr>
<tr>
<td>W</td>
<td>Window size</td>
</tr>
<tr>
<td>S</td>
<td>Negative sampling size</td>
</tr>
<tr>
<td>V</td>
<td>Number of vertices in the network</td>
</tr>
<tr>
<td>F</td>
<td>Feature size</td>
</tr>
</tbody>
</table>

Table 1: Notations used for Efficiency Study

Recall the network skip-gram algorithm, we can analyze complexity of random access with the above notations above.

\[
O(L, N, W, S) = O(LN(1 + W(1 + S)))
= O(LNW)S
\] (13)

With this baseline, we will see how can we push the boundary.

3.2 Strategies

3.2.1 GEMM

Inspired by Shihao et.al. work[5], we shared negative examples for the same context vertex and grouped update for source vertices within the same window. To denote shared examples and grouped source vertices, let’s have following notation
Table 2: Notations used for GEMM Network Skip-gram

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>{u, k_1, \ldots, k_S}</td>
</tr>
<tr>
<td>V</td>
<td>Window(u)</td>
</tr>
<tr>
<td>I_u</td>
<td>{1, 0_1, \ldots, 0_S}</td>
</tr>
<tr>
<td>L</td>
<td>I_u - \sigma(W'_U W'_V)</td>
</tr>
</tbody>
</table>

Note that U is the indices for the context vector u and negative examples, V is the indices for vectors of source vertices lies in the same window centered at the context vector u, I is the indicator matrix of dimension W x (S + 1) and L is the loss. And now recall Eq 6, we can leverage it into the following form

\[
\begin{align*}
W'_U^{(new)} &= W'_U^{(old)} + \alpha L I_U W'_V^{(old)} \\
W'_V^{(new)} &= W'_V^{(old)} + \alpha L^T W'_U^{(old)}
\end{align*}
\]

Compared with matrix-vector we used in Eq 6, it is of General Matrix Multiply (GEMM) form and hence can be implemented effectively using L3 BLAS operations. Therefore we have the GEMM Network Skip-gram

\textbf{Algorithm 5} GEMM Network Skip-gram

\begin{verbatim}
function GEMMNetskipGram(RandomWalks)
    W ← RandomInit()
    W' ← 0
    for randomWalk ← RandomWalks do
        for context ← randomWalk do
            W, W' ← GEMMUpdate(context, Window(context, randomWalk))
        end for
    end for
    return W, W'
end function
\end{verbatim}

And the complexity of random-access is now reduced to

\[O(L, N, W, S) = O(LN(W + S))\] (15)

Combined these two advantages, experiments showed that GEMM method can improve the performance effectively.

3.3 GEMM Plus

From sections 2.1 and 3, we conclude that the network skip-gram is nothing but first generating all (source, context) pairs from random-walks and then update W' for context and W for source. From section 3.2.1, we further find that efficiency improvement can be achieved by grouping source vertices within the same window centered at context vertex and share negative examples when training the ([source], context) pair. Notice that in GEMM method, we train an array of source vertices, instead of a single source vertex, against the context vertex.

If we analyze the space consumption of GEMM version of network skip-gram we will find that it requires only

\[O(S \times F + S \times W + W \times F)\]

additional spaces per-thread. With usual settings of S, W, F, it requires about 16KB per thread and with 8 threads the total consumption is just 128KB. Compared with a typical 6MB L3 cache, it clearly has not fully utilized the cache. After analyzing some typical datasets, we found that given a context vector, \(|\{v : \forall v \in V, (v, context) \in \text{RandomWalks}\}| is typically greater than W. Therefore, we could group all
these source vertices together as what we did in the GEMM method while the only difference is that in GEMM Plus, we have more source vertices grouped together and hence reduce memory access.

However, it is common that a (source, context) pair occurs multiple times when being generated from random-walks. To address the issue, we also count occurrences and adjust learning rate accordingly to simulate the original algorithm. Putting it all together, we define the GEMM Plus method.

Algorithm 6 GEMM Plus Network Skip-gram

function GEMMPlusNetworkSkipGram(RandomWalks)
    W ← RandomInit()
    W′ ← 0
    [(context, sources, occurrences)] ← Count(RandomWalks)
    for (context, sources, occurrences) ← [(context, sources, occurrences)] do
        W, W′ ← GEMMPlusUpdate(context, sources, occurrences)
    end for
    return W, W'
end function

Notice that Count(RandomWalks) will return a list of triplet where context is the index of the context vertex, sources are array of source vertices indicies and occurrences are array of occurrences of the (source, context) pairs. For example, sources_i is the index of i-th vertex which has context as its context and (sources_i, context) has appeared occurrences_i times.

To accomplish GEMM Plus update, we first index W by sources and W’ by context and negative sampling and the reset is the same as GEMM update except alpha will be adjusted by occurrences. A simple form of adjustment can be

\[ \alpha_i = \frac{\text{occurrences}_i}{\max(\text{occurrence}) - \min(\text{occurrence})} \cdot \alpha \]  

(17)

It means for the i-th source vector we scale the original learning rate by its occurrence.

Compared with GEMM version, this is effective since accesses to duplicated (context, source) pair are removed and the amortized access complexity is

\[ \Theta(V(S + \bar{d})) \]  

(18)

where \( \bar{d} \) is average degree of vertices in the network.

3.4 Experiments

We presented benchmarks result of different strategies
Note that we verified the result by classification task on the DBLP dataset.

<table>
<thead>
<tr>
<th>Method</th>
<th>Avg Precision</th>
<th>Avg Recall</th>
<th>Avg F1-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenNE</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
</tr>
<tr>
<td>Fast2Vec-Naive</td>
<td>0.83</td>
<td>0.82</td>
<td>0.82</td>
</tr>
<tr>
<td>Fast2Vec-GEMM</td>
<td>0.81</td>
<td>0.80</td>
<td>0.80</td>
</tr>
</tbody>
</table>

4 Milestones and Future Work

Current progress and outcomes can be summarized as

1. Implemented prototype for proof of the concept with naive multi-threading parallelism.
2. Added experiments pipeline for multi-class classification task, data visualization and link prediction.
3. Added re-group strategy. It showed little improvement but suffered from unbalanced task distribution.
4. Added GPU support. However, with naive implementation it suffered a lot from bank conflict so has been removed after several experiments.
5. Added GEMM support.
6. Conducted experiments on DBLP, ACM and Yago dataset.

At this phase, our major focus is to establish theoretical framework for random-walk based network embedding and preliminary implementation. Experiments have shown that under this framework not only we can develop new algorithms easily but also leverage existing skip-gram techniques to improve network embedding. At the next phase, we will try to package more existing algorithms and clarify the interfaces for extension. In addition, we will conduct extensive experiments on multiple datasets to consolidate our result. To summary, future works are listed below

1. Include Node2Vec and Metapath2Vec algorithms.
2. Clarify interface for extension.
3. Apply Fast2Vec to real-life application.
4. Extensive experiments on server.
5. Implement and test GEMM Plus.

References


