Fast2Vec: an Efficient Random-walk Based Network Embedding Framework

MA Rutian

April 16, 2019
Acknowledgment

I am proudly supervised by Dr. Reynold C.K. Cheng from the Department of Computer Science, the University of Hong Kong. He has provided continuous support and supportive environment of study. His broadened knowledge and endless passion inspired my study on this field. I would also like to express my gratefulness to Mr. Zhipeng Huang, a PhD candidate in Dr. Reynold’s group. He has mentored me and provided detailed guideline on my study. He showcased the life an outstanding researcher and always acted like my role model. In addition, Dr. Loc, post doc fellow in Dr. Reynold’s group, has provided some advices on my study.

The study has been supported by the Undergraduate Research Fellowship Program (URFP) and I would like to express my gratefulness for their work and support.
# Contents

1 Introduction 3

2 Random-walk Based Network Embedding 4
   2.1 Problem Definition .................................................. 4
   2.2 Word2Vec and Network Embedding .................................. 4
      2.2.1 Skip-Gram Model .................................................. 4
      2.2.2 Relation between Text and Network ............................. 6
      2.2.3 Random-walks Generation ....................................... 7
   2.3 Unified Framework for Network Embedding .......................... 8
      2.3.1 DeepWalk ........................................................... 8
      2.3.2 Node2Vec .......................................................... 9
      2.3.3 Metapath2Vec ..................................................... 9

3 Fast2Vec: a Novel Network Embedding Framework and its Implementation 10
   3.1 Architecture Overview ................................................ 10
      3.1.1 Scope of the Fast2Vec .......................................... 10
      3.1.2 Code Structure ................................................... 11
      3.1.3 Use Cases .......................................................... 11
   3.2 Software Design Decisions ............................................ 12
      3.2.1 Choices of Programming Languages .............................. 12
      3.2.2 Choices of External Libraries and Tools ...................... 14
      3.2.3 Design Patterns .................................................. 16
   3.3 Efficiency Study on Shared Memory .................................. 16
      3.3.1 Motivation .......................................................... 17
      3.3.2 Strategies .......................................................... 17

4 Experiments 20
   4.1 Experiment Setup ...................................................... 20
      4.1.1 Datasets ............................................................ 20
      4.1.2 Applications ....................................................... 20
      4.1.3 Metrics .............................................................. 20
   4.2 Verification of Implementation ...................................... 20
   4.3 Efficiency Study ....................................................... 21
   4.4 Interesting Properties of Network Embedding Result ............... 22
      4.4.1 Change of Test Ratio for Classification Problem .............. 22
      4.4.2 Change of Test Ratio for Link Prediction ...................... 23
      4.4.3 Change of Link Removal Ratio for Link Prediction ............ 24
   4.5 Visualization ........................................................... 25

5 Milestones and Future Work 27
Chapter 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[3]. 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[5], 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[6] 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 [15] paper which cleverly borrowed idea from word2vec[13], 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.
Chapter 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 \textit{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

$$\max_{v \in V} \sum \log Pr(N(v)|f(v))$$ (2.1)

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

2.2 Word2Vec and Network Embedding

\textit{Word2Vec} is a noble natural language processing algorithm to find vector representation of word in text data\cite{13}. And \textit{DeepWalk} is the first algorithm leveraging it to the network embedding problem. The rationale behind is the power law\cite{15}. 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 \textit{skip-gram}, one variance of \textit{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, \textit{DeepWalk} modified it to work with network data.

2.2.1 Skip-Gram Model

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

To evaluate $Pr(N(v)|f(v))$ we use conditional probability and assume all the neighbors are distributed independently so Eq-2.1 can be converted to
\[
\max_f \sum_{v \in V} \log \Pr(N(v) | f(v)) = \max_f \sum_{v \in V} \log \Pr(u_1 \ldots | v) \\
= \max_f \sum_{v \in V} \log( \prod_{u \in N(v)} \Pr(u | v)) \\
= \max_f \sum_{v \in V} \sum_{u \in N(v)} \log \Pr(u | v) 
\]

(2.2)

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 \textit{word2vec} also requires a hidden layer, we will have \( f' \) and \( W' \) to represent them accordingly.

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

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

(2.3)

However, it is expensive to evaluate the denominator of softmax function so Mikolov etal. [13] proposed negative sampling to simulate \( \log \Pr(u | v) \)

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

(2.4)

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 2.4

\[
\frac{\partial}{\partial W'_u} = \frac{1}{\sigma(W'_u \cdot W_v)} \cdot \frac{\partial \sigma(W'_u \cdot W_v)}{\partial W'_u} \\
= \frac{1}{\sigma(W'_u \cdot W_v)} \cdot \sigma(W'_u \cdot W_v) \cdot (1 - \sigma(W'_u \cdot W_v)) \cdot \frac{\partial W'_u \cdot W_v}{W'_u} \\
= (1 - \sigma(W'_u \cdot W_v)) \cdot W_v \\
\frac{\partial}{\partial W'_k} = \frac{1}{\sigma(-W'_k \cdot W_v)} \cdot \frac{\partial \sigma(-W'_k \cdot W_v)}{\partial W'_k} \\
= \frac{1}{\sigma(-W'_k \cdot W_v)} \cdot \sigma(-W'_k \cdot W_v) \cdot (1 - \sigma(-W'_k \cdot W_v)) \cdot \frac{\partial -W'_k \cdot W_v}{W'_k} \\
= (1 - \sigma(-W'_k \cdot W_v)) \cdot (-W_v) \\
= -\sigma(W'_k \cdot W_v) \cdot W_v \\
\frac{\partial}{\partial W_v} = (1 - \sigma(W'_u \cdot W_v)) \cdot W'_u - \sum_k \sigma(W'_k \cdot W_v) \cdot W_k 
\]

With the equation, we will be able to derive the update function for both \( W \) and \( W' \).
\[ W_v = W_v + \alpha \cdot \frac{\partial}{\partial W_v} \]
\[ = W_v + \alpha \cdot \{(1 - \sigma(W'_u \cdot W_v)) \cdot W'_u - \sum_k \sigma(W'_k \cdot W_v) \cdot W_k \} \]

\[ W'_v = W'_v + \alpha \cdot \frac{\partial}{\partial W'_v} \]
\[ = W'_v + \alpha \cdot \{(1 - \sigma(W'_u \cdot W_v)) \cdot W_v \} \]

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

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

**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 ← Wsource
                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 [13, 10] 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 [15], 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 [15] work, they proposed to use random-walk as a substitute of sentences in text. For example, a sentence in word2vec may looks like

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

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 \textit{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

\[
\sum_{u \in \text{Neighbor}(v)} Pr(u|v) = 1 \tag{2.7}
\]

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

Note that \textit{Draw(current, Network)} choose a neighbor vertex of \textit{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 \textit{RandomWalks} we will be able to train our network with skip-gram model 1 with just a little modification.
Algorithm 3 Network Skip-gram

```python
function NetworkSkipGram(RandomWalks)
    W ← RandomInit()
    W' ← 0
    for randomWalk ← RandomWalks do
        for context ← randomWalk do
            for source ← Window(context, randomWalk) 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
```

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

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

```python
function NetworkEmbedding(Network)
    RandomWalks ← RandomWalksGeneration(Network)
    W, W' ← NetworkSkipGram(RandomWalks)
    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 [16], faster softmax computation [21, 8] and learning rate adjustment [11]. 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.

2.3.1 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

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

(2.8)

Note that \( v \) is the current vertex and \( u \) is one of the neighbors of \( v \), which makes it one of the candidates of the next vertex.

### 2.3.2 Node2Vec

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

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

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

(2.9)

Note that \( t \) represents the vector we were before the current vertex \( v \) and \( \pi(v|u) \) 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

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

(2.10)

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.

### 2.3.3 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.

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

(2.11)

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

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

(2.12)

where \( \phi(v,x) \) is the type of the edge connecting \( v \) and \( x \).
Chapter 3

Fast2Vec: a Novel Network Embedding Framework and its Implementation

As discussed in previous chapter, we can summaries a class of random-walk based network embedding under a unified theoretical framework and hence rises the challenge of designing and then implementing a software package to bring the novel framework into real world.

There are two major objectives when it comes into the implementation of Fast2Vec. First, we want to provide a high level of extensibility and flexibility for user who wants to compose application with network embedding and developer who wants to investigate into new random-walk embedding algorithm. To achieve the purpose, we want to minimize the code needed to implement new algorithm while maximize the reusable code among different scenarios. In addition, we also want to provide certain level of abstraction so that the program interface won’t be affected by the real implementation and hence developer optimize certain section of code without contaminating other’s code. Second, we want to provide an implementation of framework with optimized performance. Current available network embedding packages like OpenNE are mostly implemented in Python and written for demo purpose only. Although the underling computation libraries are NumPy\[20\], SciPy\[9\] and Sci-Kit Learn, we are still able to find considerable room for performance improvement.

In the rest of this chapter, all the efforts to achieve these goals will be addressed in a top-down fashion. First, we will provide an architecture overview of the Fast2Vec framework, which describes the major modules and their respected functions. Second, we will explore important software design decisions including choices of languages, libraries and design patterns. Lastly, we will take a closer look at how do we optimize the training time of network embedding in a shared memory environment.

3.1 Architecture Overview

3.1.1 Scope of the Fast2Vec

No matter you are a user looking for an out-of-the-box network embedding program to process your network data, or a developer wanting to design and explore your novel network embedding algorithm, you will always find the Fast2Vec helpful since it can serve you as

- **a Framework**: If you have brilliant idea on certain aspect of network embedding, like new way of random-walk generation or better training algorithm, then you only need to express the core of your idea, and all the other dirty work like loading a graph and process program arguments will be taken care by the Fast2Vec.
• **a Program:** If you are a user who only needs a tool to process your network data and then build fancy applications, the Fast2Vec also provides you an executable to run. It will read your instruction, load your data, generate random-walks, train the model and finally materialized it. The executable is optimized and extensive so you can leverage state-of-the-art algorithms like DeepWalk, Node2Vec and Metapath2Vec for your work at ease. To try multiple combinations of algorithms or hyper parameters, you only need to modify few lines in the configuration file without touching the code. Better still, the Fast2Vec also comes with some scripts for popular application like link prediction and classification.

• **Libraries:** The Fast2Vec consists of independent libraries for mathematical computation, graph processing, random-walk generation and training so if you want integrate any of the work above, you can simply include it into your program.

### 3.1.2 Code Structure

To provide the functionality in 3.1.1 and comply with some of the best practice in software engineering, we structured the Fast2Vec in a layered fashion and the details and rationales will be discussed below.

As stated in previous chapter2, a random-walk based network embedding algorithm can be abstracted into two phases, random-walks generation and training. Consequently, it is natural to separate implementation of these two phases. In the Fast2Vec implementation, we actually chunk the network embedding task into four ordered layers.

1. **Graph Layer:** we provide a graph layer for user to load and manipulate input data.

2. **Random-walk Generation Layer:** the loaded graph can be pumped into random-walk generation layer. In this layer, user will build a random-walker from the factory method to generate random-walks from the graph. Particularly, we implemented three classes of random-walk algorithms in the Fast2Vec, namely DeepWalk, Node2Vec and Metapath2Vec.

3. **Training Layer:** the generated random-walks will then be supplied into the training layer, and after training the model will be materialized into files.

4. **Application Layer:** After building the model, user can feed it into application layer for some out-of-the-box application experience. In the Fast2Vec, we have provided scripts for classification and link prediction.

When developing a network embedding algorithm, each of these layers are coordinated by abstract programming interfaces so the implementation details are hidden behind each layer. After compilation of the whole Fast2Vec program, each of these layers are then coordinated by the input configuration file. User can then tunes the random-walk generation and training algorithm through a set of compilation.

### 3.1.3 Use Cases

By decoupling random-walk generation and training phases into three layers, user can explore multiple combination of random-walk generation algorithms and training algorithms to find a better fit, and developer can optimize towards only a minimal area of the code. Further more, such an architecture enable finer grained of inspection during the execution. It is possible to verify the effectiveness and benchmark the efficiency of each layer with this architecture and below are just several use cases we found useful.

• **Develop New Random-walk Algorithm:** As stated in previous chapter, a new random-walk based network embedding algorithm is usually determined by its random-walk generation algorithm and particularly determined by the transmission probability. In the Fast2Vec, developer saves time by reuse the provided graph layer and training layer, and only needs to pay attention to update the random-walk generation layer.

• **Develop New Training Algorithm:** Developer might also want to optimize training algorithm, and he only needs to touches a single piece of code a time thanks to the layered approach.
Benchmark Random-walk Overhead: There are several algorithms involve sophisticated random-walk scheme[5] and hence we want to verify the fact not only from a theoretical point of view but also from empirical study.

Unit Testing: As mentioned above, a developer can extend and modify every layer of the Fast2Vec. With our layered approach, there is no need to rerun the whole program for the verification, and the developer can simply extracts the layer out, feeds it with certain input and see if the output matches the expected output.

Minimize Redundant Computation: In a monadic architecture, we need to rerun the whole program to try a different set of hyper parameters, like feature size, number of epochs and number of negative examples, while in the Fast2Vec, we only need materialize the generated random-walks and rerun the computation phase. This is especially useful when the random-walk generation phase is time consuming.

Downstream Applications: The Fast2Vec, as a program, also allows user to explore some out-of-the-box downstream applications of network embedding, like classification and link-prediction.

In conclusion, the layered architecture, which was implemented in the Fast2Vec, is the major effort to achieve better flexibility and efficiency than the other packages and framework.

3.2 Software Design Decisions

After an architecture overview, we now take a closer look at the implementation details of the Fast2Vec. The rest of this section will first describe the programming languages chosen in the Fast2Vec, then the libraries leveraged and finally some design patterns techniques we used to further improve the quality of code.

3.2.1 Choices of Programming Languages

As described in the 3.1.3, the Fast2Vec provides not only network embedding computation but also some downstream applications. To deal with the huge volume of data and proceed with most kinds of complex algorithm, the computation part, the graph layer, the random-walk generation layer and the training layer, of the Fast2Vec is more performance oriented. On the other hand, users might need various applications to best fit their requirements and therefore the application layer of the Fast2Vec is more flexibility oriented.

To entertain different requirements, layers are implemented in different languages. On the performance side, static language is chosen for better performance and type-checking while on the application side, scripting language is chosen to provide better flexibility and ease to write code.

Static Programming Language

Most of the main stream static programming languages provide a highly optimized compiler, prosperous ecosystem and expressive yet safe syntax. The major candidates among them are Java, C++ and C[19] and we chose C++ instead of Java or C for the following reasons.

- Low Level Control: In our efficiency study of network embedding, we considered as low level control of the programming execution as memory layout and cache behavior. In Java, a language running on Java Virtue Machine, it takes significant greater effort to have this level of fine tunning of the program while still have no guarantee on the program behavior. To ensure the expected outcome, C/C++ becomes our choices.

- Zero Overhead Abstraction: Again in our efficiency study experiment, it is critical to ensure behavior of the program since any unexpected overhead may misleads our analysis. From this perspective, C/C++ both provide the great feature of zero overhead abstraction. That is to say, we won’t pay any computation resources for abstractions we haven’t made or functions we haven’t asked. Such a purity will increase our confidence towards the experiments result.
• **Object Oriented Programming**: To achieve the layered architecture and extensibility of the code, the famous object oriented programming feature is required. Therefore, we have to rule out C although it fits all the requirements above. It takes considerable effort to achieve a similar effort in C than in C++ or Java.

• **Interface with Graphics Processing Unit**: In the initial stage of the development of the Fast2Vec, we planned to accelerate the computation therefore we have to ruled out Java. Since GPU interface from Java is not as matured as the one on C/C++ [22].

   Although one may contradict to the choice of C++ rather than Java for the following reasons, we found either alternatives or solutions to mitigate the short come of C++.

• **Distributed Computing**: Since the trending Spark is the premium platform for distributed computing, one may argue that a JVM compatible programming language is a better fit for such a big data analysis task as network embedding. Nevertheless, scholars have shown that the distributed library on C++ like OpenMPI provides no worse performance than Spark[18]. In addition, we also found that memory on a single machine nowadays is capable of most of network data. Therefore, it is less important to distribute the tasks to multiple machines consider the network transmission overhead.

• **Memory Management**: JVM offers garbage collection (GC) system while C++ not and therefore developer has to manage it manually. Therefore, one may argue that Java is a better choice in such a large scale project to reduce the risk of memory leak. However, C++ provides smart pointer mechanism, which acts like a garbage collection system and will free the memory when it comes out of the scope. In addition, we also found that the JVM GC comes at a price since it will brings memory management overhead and unexpected system halt.

With the reasons above, we concluded that C++ is a better choice over Java and C for the graph layer, random-walk generation layer and the training layer for the Fast2Vec.

**Dynamic Scripting Language**

In the application layer, we focus less on how fast a computation can be carried out while more on how easy it is for a user or developer to prototype new awesome applications. Therefore, a dynamic typing script language is a better fit for such an agile development style.

Different from the arena of static languages, choice of script language is fairly straight forward since Python is the premium choice for the following reasons.

• **Elegant and Expressive Syntax**: Python’s syntax is well known to be handy and user can express their idea in several simple lines of code. The workload in development is lower compared with working with C++ or even other script languages like Shell Script.

• **Powerful Libraries**: Popular AI-driven applications and functions have been built into Python’s ecosystem and most of them rely on efficient code written in C, and therefore keep a good balance between ease of use and efficiency.[14]

• **Strong Machine Learning Community**: As stated in 1, the primary goal for network embedding is to mitigate the representation gap between network data and machine learning algorithms. Since Python enjoys a better machine learning community than the other candidates, we might better integrate other’s advancements and contribute to the community by using Python to process our embedding outcomes.

Therefore, the application layer of the Fast2Vec is implemented in Python. It provides an array of applications including classification, link-prediction and data visualization.
3.2.2 Choices of External Libraries and Tools

After deciding the choice of languages, we were confronted with the problem of choosing related libraries and tools. To avoid redundant code and leverage as much research outcomes as possible, we have spent great effort in selecting and combining the libraries and tools.

In the rest of this subsection, we are going to discuss the major Libraries and tools we used in the Fast2Vec, and the rationales behind contradicting to other competitors.

Dependency Control and Compilation: CMake

All mainstream programming languages are accompanied with code management tools to ensure correct dependency control, and in the world of C++, we also want to ensure correct compilation configuration. There are two basic tools to achieve the goals: CMake and Make. The basic function of these tools are similar. User writes configuration file and the tool will generate corresponding instructions to compile the program, but we finally chosen the first one contradicting to the latter for the following reasons.

- **Automatic Library Locating and Configuration**: As will be mentioned below, this project involves considerable amount of libraries and we want our users and developers rest at ease to configure all these dependencies. In Make, you need to manually input the paths of libraries and configure compilation parameters accordingly. In comparison, all you need to do is to instruct CMake to find and configure certain libraries.

- **Target Oriented Compilation**: In CMake, the code is structured into multiple targets, and an executable can be compiled from source code and additional targets. When adding a target to another target, the CMake will figure out the dependency and hence avoid error pruning manual dependency control. In addition, CMake also provides a tree-like structure for the project, which is absent in Make.

- **Better IDE Support**: With the target oriented information, it becomes easier for IDE to figure out the structure and context of the code, and therefore provide better support, including auto-completion and smart correction.

- **Cross Platform Support**: Make is a tool exclusive on unix-like system while CMake is a cross platform to. That is to say future users and developers can continue their works with the Fast2Vec on their Windows machine without painful migration process.

- **User Friendly Design**: Make simply figures out the dependencies and then compiles the program while CMake provides heartful colorful report about the progress, error and warning. This help user to locate issues.

With all these desirable features, we chosen CMake as the tool to manage our code and the compilation process.

High Thread-level Performance Parallelism: Thread Building Block (TBB)

As described in 2, the training phase can be trivially parallelized into multiple groups of random-walks. Then we assign these groups to the worker threads, and each thread just update the model according to the group of random-walks given. Although there is theoretical possibility of data race issues, experiments have shown that concurrent update on the model won’t effect the accuracy of network embedding in a significant level [15, 7].

Therefore, the most trivial implementation of parallelism in training phase is to spawn a pool of threads and then allocate group of random-walks to process. Such a producer-consumer model has been thoroughly studied and we can simply leverage the research outcome[17] to boost performance of the Fast2Vec.

To implement the parallelism, we can opts from native POSIX library[2], OpenMP[4], and the Thread Building Block (TBB)[17], and we made the decision for the following reasons.
• **Generic Parallelism Description**: To parallelize a range of tasks, all the other libraries require developers to explicitly write a loop, distribute task, manage execution and control the shared resources. However, TBB provides a very generic view towards parallelism. All the developers need to do is to write a class that will be instantiated with the shared resources and called to process a range of tasks.

• **Smart Job Allocation**: In some circumstances, it is not enough to equally split the tasks to all the threads since workload for task varies against each others. Fortunately, the TBB and its runtime is aware of this issue and can automatically balance the workload. If you want to have a fine-grained control, you may also write your own class to split the tasks.

• **Optimized Performance**: Developed by Intel and continuously improved by the community, TBB is proven to be more efficient[17] than the traditional libraries.

Therefore, parallelism by TBB is one of the options in the Fast2Vec against traditional thread level parallelism.

**Linear Algebra Computation: Math Kernel Library (MKL)**

As described in 2, the training phase requires considerable amount of computations, and most of which are linear algebra operations. Since linear algebra operations are core for most scientific computation, research community have built tremendous amount of works to optimize such a class of operations. Therefore, we decided to reuse other’s work instead of writing our own mathematical library.

Most of the linear algebra libraries provide similar functionalities and abstraction. We chose MKL for its outstanding performance and its coordination with TBB, another Intel library TBB in our cart.

**Acceleration with Graph Processing Unit (GPU): CUDA**

Despite efficient computation on CPU, we also wanted to leverage the computation power of Graph Processing Unit (GPU). Therefore, we have also used CUDA to interface with the GPU and carry out computation.

Due to the hardware constraints and availability of the library, CUDA is the natural choice over products from its counter-part, AMD.

**Graph and Auxiliary Support: Boost**

After deciding the libraries for computation, we also wanted to find better support for processing data and arguments. Since C++ stuck to its zero overhead promise, it provides very limited built-in libraries, and it was a tedious task to implement everything from scratch.

Fortunately, Boost, a collection of libraries to support many day to day routines, came to rescue from the following perspectives.

• **Efficient yet Configurable Graph Library**: It is common practice to store a graph into either adjacent list, adjacent matrix or edge list. However it is hard to do it right. Boost provides a well-written graph library so that we could rest assured that the network data has been processed and stored correctly. Better still, the library is highly customizable so that we can consolidate both heterogeneous information network and homogeneous information network under a unified interface.

• **Program Parameters Support**: As mentioned in 3.1, we have four layers in the Fast2Vec and each of them requires a set of parameters. Boost supports parsing, checking and storing parameters of different types from multiple sources. Compared with implementing the program parameters parser all from scratch, it is a better choice to use such a tested piece of work.

Although Boost’s functions is not limited to what were mentioned above, they are the two major use cases of Boost in the Fast2Vec project.
3.2.3 Design Patterns

After deciding the overall architecture, programming languages and libraries, it comes to detailed implementation. In this sub section, we will explore the design patterns, novel software engineering ideas, that have been practiced in the Fast2Vec.

Dependency Injection

To implement layered architecture and hide implementation detail behind, we used the dependency injection (DI) techniques.

That is to say, the graph layer of the Fast2Vec only represents the interface of operations related to a graph while the real implementation, Boost’s Graph, was injected into the Fast2Vec graph. Similarly, we implemented wrapper functions for mathematical computation and then inject the MKL into them.

By doing so, we separate representations and implementations. Subsequently, developer will find it easier to replace and update the implementation. For example, if one has optimized the general matrix multiplication (GEMM) operation, he simply replaces the MKL implementation with his own, and there is no need to modify and test other parts of the code since the interface should remains the same.

In conclusion, DI is a technique used in the Fast2Vec to reduce coupling and increase coherence of the code.

Factory Model

Another challenge in the implementation of the Fast2Vec is to ensure correctness when instantiates each layer.

Since we provided different classes of implementations of each layer, it is critical to ensure that we have instantiate the right one correctly. Due to the constraint of C++, there is no static-checking mechanism for polymorphism of classes.

Therefore, we deployed factory model as a centralized way to ensure the rule of object instantiation. Taking random-walker’s instantiation as an example. First user supplies parameters to the random-walker factory and then the factory reads the parameters to decide which random-walker to instantiate.

Although such a design requires developers to modify the factory method when develop new random-walker, it will help developers to put all the rules at one place and it is some of the best practices to ensure the correctness of the program.

Inversion of Control

Unlike pure layered architecture, different layers in the Fast2Vec requires information not only from layer one level above it but also additional information from other layers.

To avoid writing a cumbersome class to manage all the information, the Fast2Vec used inversion of control approach to tackle the problem. That is to say, manager of information will release APIs for other to access and manipulate its own information. For example, a graph object can not only be used by the random-walk generation phase but also later used in the training phase.

By distributing the data accesses to multiple small and manageable classes, we ensure more readable code for developers.

3.3 Efficiency Study on Shared Memory

As proved empirically in the word2vec paper[13], 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.
3.3.1 Motivation

The *skip-gram* model with stochastic gradient descent[16] is trivially parallelizable but will suffer a lot from the cache miss and ping-pong update. In addition, the original version of *network skip-gram* 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 3.1: Notations used for Efficiency Study

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

\[
O(L, N, W, S) = O(LN(1 + W(1 + S))) = O(LNW\ S) \tag{3.1}
\]

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

3.3.2 Strategies

GEMM

Inspired by Shihao et.al. work[8], 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>
<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>|_U - \sigma(W_U^TW_V^T)</td>
</tr>
</tbody>
</table>

Table 3.2: Notations used for GEMM Network Skip-gram

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 \(W \times (S + 1)\) and \(L\) is the loss. And now recall Eq 2.6, we can leverage it into the following form

\[
W_U^{(new)} = W_U^{(old)} + \alpha LW_V^{(old)} \\
W_V^{(new)} = W_V^{(old)} + \alpha L^TW_U^{(old)} \tag{3.2}
\]

Compared with matrix-vector we used in Eq 2.6, 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*
Algorithm 5 GEMM Network Skip-gram

<table>
<thead>
<tr>
<th>function GEMMNetworkSkipGram(RandomWalks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W \leftarrow$ RandomInit()</td>
</tr>
<tr>
<td>$W' \leftarrow 0$</td>
</tr>
<tr>
<td>for randomWalk $\leftarrow$ RandomWalks do</td>
</tr>
<tr>
<td>for context $\leftarrow$ randomWalk do</td>
</tr>
<tr>
<td>$W, W' \leftarrow GEMMUpdate(context, Window(context, randomWalk))$</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>return $W, W'$</td>
</tr>
<tr>
<td>end function</td>
</tr>
</tbody>
</table>

And the complexity of random-access is now reduced to

$$O(L, N, W, S) = O(LN(W + S)) \quad (3.3)$$

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

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 3.3.2, 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

$$O(S \times F + S \times W + W \times F) \quad (3.4)$$

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, $|\{v : \forall v \in V, (v, context) \in RandomWalks\}|$ 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 time 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

<table>
<thead>
<tr>
<th>function GEMMPlusNetworkSkipGram(RandomWalks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W \leftarrow$ RandomInit()</td>
</tr>
<tr>
<td>$W' \leftarrow 0$</td>
</tr>
<tr>
<td>$[(context, sources, occurrences)] \leftarrow Count(RandomWalks)$</td>
</tr>
<tr>
<td>for $(context, sources, occurrences) \leftarrow [(context, sources, occurrences)]$ do</td>
</tr>
<tr>
<td>$W, W' \leftarrow GEMMPlusUpdate(context, sources, occurrences)$</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>return $W, W'$</td>
</tr>
<tr>
<td>end function</td>
</tr>
</tbody>
</table>

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.

**Choices of Adjustment Functions**

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.

Let’s denote α_ij as the learning rate when training with i-th vector as context vector and j-th vector as source vector, occ_ij as the number of occurrences of (i, j) pair occurred among the random-walks and α as the initial learning rate.

The most naive form of learning rate adjustment can be

\[ α_{ij} = α \]

That is to say we simply ignore the information of number of occurrences. If we want to leverage this piece of information, the easiest way to do so is to multiply the original learning rate by number of occurrences

\[ α_{ij} = occ_{ij} \cdot α \]

However, it will lead to divergence when we encounter a popular pairs of vectors. To mitigate the problem, we also tried several formula to constraints the range of adjusting factor to be [0, 1].

The first trail is to use self-adjustment

\[ α_{ij} = \frac{occ_{ij}}{1 + occ_{ij}} \cdot α \]

If we want to consider all possible sources vectors with context i, we might also want to try.

\[ α_{ij} = \frac{occ_{ij}}{\sum_j occ_{ij}} \cdot α \]

If we want to be inspired by the ReLU then we could also use

\[ α_{ij} = \min(occ_{ij}, c) \cdot α \]

where c is a user defined hyper parameter for the maximum of the adjusting factor.

Surprisingly, later experiments shown that the most naive implementation brings the highest accuracy.

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

\[ Θ(V(S + d)) \]

where d is average degree of vertices in the network.
Chapter 4

Experiments

The objectives of this chapter is to

- **Verify the Fast2Vec Implementation**: To ensure the correctness of our implementation, we want to plug the embedding result into several downstream applications.

- **Explore Properties of Network Embedding Algorithms**: With such a sharp tool to do network embedding, we are curious

- **Benchmark**: Since the primary goal of the Fast2Vec is to improve performance, we want to verify our achievement by benchmarking on several datasets.

4.1 Experiment Setup

4.1.1 Datasets

- **DBLP**: a scholarship dataset consists of 51377 edges, 15648 vertices, 8 kinds of edges and 4 kinds of vertices.

- **YAGO**: a knowledge base consists of 2137469 vertices and 4431511 edges.

4.1.2 Applications

- **Multi-class Classification**: Given a vector representation of a vertex in the network, we want to determine what kind of vertex it is.

- **Link Prediction**: Given two vector representation, we want to determine whether they are connected together and connection is defined by whether there exist a path between two vertices.

4.1.3 Metrics

- **Precision**: Number of true positives divided by the number of samples.

- **Recall**: Number of true positives divided by the number of reported positives.

- **F1-Score**: Harmonic mean of precision and recall.

4.2 Verification of Implementation

First, we verified the result by classification task on the DBLP dataset.
### Table 4.1: Classification Result 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>

### 4.3 Efficiency Study

We have run experiments for both DeepWalk and Node2Vec algorithms on the DBLP datasets.

Figure 4.1: Runtime of DeepWalk on the DBLP dataset

They have shown that naive implementation already yield satisfactory performance and scalability.
Better still, two acceleration strategies achieve minimal computation time even with very little number of threads.

4.4 Interesting Properties of Network Embedding Result

We are also interested in the performance of network embedding result. Use our network embedding result will yield very high accuracy even with limited number of ground truth.

4.4.1 Change of Test Ratio for Classification Problem

Let’s denote the ratio of ground truth label for test against training as test ratio and we reported the variation of metrics with respect to the change of test ratio.

![Figure 4.3: Classification Metrics Change w.r.t to Test Ratio](image)

According to the figures above, we found that the accuracy remains at a very high level even with limited...
ground truth provided. This yield a theory that the network embedding result can be used to classification problem even when the dataset comes with a small number of ground truth.

4.4.2 Change of Test Ratio for Link Prediction

We also want to examine whether the property mentioned above is applicable for other down stream applications. In this subsection, we will present the result on link prediction where we also varied the test ratio.

In the link prediction, we first selected $N$ connected pair of vertices as positive examples, and then randomly selected $N$ pair of vertices as negative examples. We splat the data according to the test ratio.

Figure 4.4: Link Prediction Metrics Change w.r.t to Test Ratio

From the figures, we can observe two interesting facts.

- **Both methods yield similar performance with different test ratios**: Again, this shows that the network embedding result requires less ground truth label when it comes to downstream application.
- **Both methods yield a recall higher than precision**: By definition, we can imply that our model is more optimistic since the false negative is greatly smaller than the false positive.

### 4.4.3 Change of Link Removal Ratio for Link Prediction

To come up with a link prediction experiment, we removed existing links between vertices according to a certain ratio. We also tested our model with different link removal ratios.

Figure 4.5: Link Prediction Metrics Change w.r.t to Link Removal Ratio

The bell shape is a common pattern for such a task since a fully-connected or a full-disconnected is easier to predict.

In conclusion of this section, we found that network embedding results have a good property that they require very little ground truth for downstream application.
4.5 Visualization

In the DBLP dataset, all the authors have been categorized into four research fields: machine learning, system, data mining and network. To present a visualization of network embedding result, we conducted a TSNE[12] reduction on those vectors, plotted them onto a 2-dimensional space and finally colored them with the labels.

Through the clear boundary between different cluster of vectors, we could tell that the network embedding results, no matter it is DeepWalk or Node2Vec, have done a great job in capturing the information of each vertex.

In Metapath2Vec, we have a hyper parameter called metapath and the plots came from embedding of different metapath. They are Paper-Author-Paper, Conference-Author-Conference and Conference-Paper-Author-Paper-Conference respectively. As we can see from the visualizations, Paper-Author-Paper and Conference-Paper-Author-Paper-Conference yielded very chaotic results and hence further implied bad quality downstream applications. From the plots, we can conclude that the Metpath2Vec is very sensible towards the hyper parameter selection.
Figure 4.8: TSNE Plots for Metapath2Vec on DBLP
Chapter 5

Milestones and Future Work

Lastly we summarize our work of Fast2Vec and its contribution from different perspectives.

For theory of network embedding, we have

- **Established a Unified Framework for Random-walk Based Embedding**: We conclude that a class of network embedding algorithms can be described by their random-walk process and uniquely determined by their transmission probability.

- **Empirically Studied the Properties of Network Embedding**: We, through series of experiments, found that network embedding result works for well downstream applications even when the number of ground truth are limited.

- **Studied Metapath2Vec Under the Change of Metapaths**: We found that the Metapath2Vec is unstable under the change of metapaths and hence yield results of different quality.

From engineering perspective, we have

- **Implemented an Efficient Network Embedding Framework**: The framework features not only in its high performance but also flexibility. With our Fast2Vec, user and developer can leverage and develop new algorithms at fast.

- **Deployed Advance Parallelism Techniques in Network Embedding**: Before the Fast2Vec, public packages solving the network embedding problem were mostly written in Python and had very limited performance and scalability. Now, with the application of fine-tuned libraries like MKL and TBB, we have achieved state-of-the-art performance on some popular algorithms like DeepWalk, Node2Vec and Metapath2Vec.

- **Tested Application of GPU in Network Embedding**: Although GPU assisted computation is a trending field of study, we found that due to the large number of random access, random-walk based network embedding is not suitable for GPU acceleration.

- **Prepared Several Datasets and Applications**: At the end of big data analysis, we ought to have the data and corresponding applications. We prepared two datasets and two applications in the Fast2Vec.

Due to the time and computation limitation, there are some work can be done in the future.

- **Distribution**: After the final documentation and refining touch, the code will be open sourced.

- **Experiments**: Experiments on more datasets with more metrics should be conducted. In addition, it is desirable to have some kind of intrinsic evaluation of the network embedding.

- **Publication**: Summarize the core contribution of the study and write a paper on it.
Bibliography


