Static Embeddings

Shoval Frydman

December 2020

Abstract

Graph embeddings are used for multiple tasks, including mainly node classification, link prediction and visualization. Current approaches suffer from three main drawbacks: a) their computing time and cost become prohibitive in very large graphs. b) They cannot handle online information that is constantly updated and c) their precision drops with the degree of the vertices. We here propose an online graph embedding method which starts with an initial embedding calculation of a fixed core of vertices with highest degree, using a known embedding algorithm, and then uses this embedding in order to calculate the whole graph embedding. The vertices are inserted one at a time to the embedding and this calculation is done with three different approaches: a) a weighted combination (OGRE), b) a directed regression (DOGRE), and c) a directed weighted regression (WOGRE). The vertices are inserted one at a time to the embedding. WE show that OGRE obtains better performance in all three tasks compared to the three standard graph embedding algorithms: Node2Vec, HOPE and Graph Factorization. Specifically, for node classification tasks, OGRE with an small initial seed of 100 nodes obtains an accuracy higher than all other algorithms by approximately 30% in the DBLP and PUBMED datasets, and can be implemented on very large graphs, such as Yelp with 700k vertices and 7M edges. Finally, OGRE can be combined with a GCN seed to apply node classification to very large graphs.

1 Introduction

1.1 Graph Embedding

Graphs are commonly used in different real-world applications, such as social and communication networks. The interest in performing machine learning on graphs is growing, e.g. prediction of new friendships in social media, classification of people by their age, gender, etc. Mathematical and statistical operations on graphs are limited and applying machine learning methods directly to graphs is challenging. In this situation, embeddings appear as a reasonable solution. Graph embeddings, as will be discussed in here, are the transformation of vertices in graphs to a vector representation in $\mathbb{R}^d$. Embedding should capture the graph topology, vertex-to-vertex relationship, and other relevant information about graphs, sub-graphs, and vertices.
In general, graph embedding is needed for few reasons:

- Graphs consist of edges and vertices that can only use a specific subset of mathematics, statistics, and machine learning, while vector spaces have a richer toolset of approaches. In addition, vector operations are simpler and faster than comparable operations on graphs.

- Adjacency matrix of graphs describes connections between vertices and its size is $N \times N$ where N is the number of vertices. When the graph is large, e.g. one million nodes, such representation is impossible to handle. Thus, embeddings are more practical since they pack vertex properties in a vector with a smaller dimension.

- In order to perform machine learning tasks such as node classification, link prediction, etc, one needs a comfortable framework as embedding.

1.2 Static Graph Embedding

Given a large graph $G = (V, E, W)$, where V is the group of vertices, E is the group of edges and W is the weighted adjacency matrix of G, a graph embedding is defined as a mapping $f : v_i \rightarrow y_i \in \mathbb{R}^d \ \forall i \in \{1, ..., |V|\}$ such that $d << |V|$ and the function f preserves some proximity measure defined on the graph G. An embedding therefore maps each vertex to a low-dimensional feature vector and tries to preserve the connection strengths between vertices. We would like to find this embedding in the shortest time and smallest memory usage possible.

There are multiple embedding methods for static graphs which do not use extra information on the graph or its vertices (such as vertex features) in order to compute the embedding [5]. They can be categorized into three broad categories:

- **Factorization based algorithms** ([14], [12], [1], [15], [13]): Such algorithms represent the connections between vertices in the form of a matrix and factorize this matrix (in several ways) to obtain the embedding.

- **Random Walk based algorithms** ([3], [6]): Random walks are used to approximate many properties in graphs such as similarity. These methods sample sequences of vertices from the graph using random walks, with the idea that vertices that are "close" in the graph connectivity will also frequently appear close in a sampling of random walks.

- **Deep Learning based algorithms** ([4], [16], [10]): These methods use deep autoencoders for dimensionality reduction due to their ability to model non-linear structure in the data. An embedding generated from a deep autoencoder can capture non-linearity in graphs.

- **GCN** ([10]): A powerful type of neural network designed to work directly on graphs and leverage their structural information.

In addition, there are multiple embedding methods which do use this extra information such as Graph Embedding of Large-scale Attributed Graphs (GLACE) [2], Neighborhood Based Node Embeddings (NBNE) [17], GraphStar [11], GraphSaint [8], SGE [7], GraphSAGE [18], NetSMF ([9]), etc. We do not compare ourselves to them because our suggested methods do not use any extra information except of the given graph itself.

An important limitation of all existing methods is their prohibitive cost in CPU and memory for large graphs. Myriad static graph embedding methods, such as [6], [13], [1], claim to run in a "reasonable" time over very large networks (millions of vertices). However, in parctice, most of these algorithms require extensive cpu and memory and do not scale linearly with the number of vertices above some minimal threshold, and as a result actually fail to work for large dense graphs, i.e. the memory they yield is too much to handle.

2 Related work

In this section we will define in details the existing embedding algorithms relevant to us.

2.1 Node2vec

Node2vec [6] is a random walk based algorithm for scalable feature learning in networks. It preserves higher-order proximity between vertices by maximizing the probability of occurrence of subsequent vertices in fixed length random walks. The approach returns feature representations that maximize the likelihood of preserving network neighborhoods of vertices in a d-dimensional feature space. They use 2nd order random walk approach to generate network neighborhoods for vertices. By choosing an appropriate notion of a neighborhood, node2vec can learn representations that organize vertices based on their network roles and/or communities they belong to. It is achieved by developing a family of biased random walks, which efficiently explore diverse neighborhoods of a given vertex.

2.2 Graph Factorization

Graph factorization [1] allows us to find latent (unobserved) factors associated with vertices which can be used for further inference in the network. It is highly desirable to solve the factorization problem suggested there efficiently.
There exists a wide range of objective functions for graph reconstruction and factorization. Many of them share the property that there are vertex attributes Z_i which can be used to determine whether an edge (i, j) should be established, and these attributes will be the vertex embedding. They restrict themselves to a simple inner product model where they assume that information regarding the presence of an edge Y_{ij} can be captured efficiently via $< Z_i, Z_j >$.

The loss function is:

$$f(Y, Z, \lambda) = \frac{1}{2} \sum_{(i,j) \in E} (Y_{ij} - < Z_i, Z_j >)^2 + \frac{\lambda}{2} \sum_i ||Z_i||^2$$

where Y_{ij} is the weight of the edge (i, j), Z is the factor (embedding) matrix with vector Z_i for vertex i and λ is a regularization parameter.

2.3 HOPE

HOPE [13] preserves higher order proximity by minimizing:

$$L = ||S - Y_S Y_t^T||_F^2$$

where S is the similarity matrix, Y_S is the source embedding and Y_t is the target embedding. The authors experimented with different similarity measures, including Katz Index, Rooted Page Rank, Common Neighbors, and Adamic-Adar score. They represented each similarity measure as

$$S = M_g^{-1} M_l$$

where both M_g and M_l are polynomial of matrices and sparse. This enables HOPE to use generalized Singular Value Decomposition (SVD) to obtain the embedding efficiently.

2.4 Graph Convolutional Network (GCN)

Given a graph $G = (V, E)$, with an adjacency matrix $A \in R^{|V|x|V|}$, a diagonal degree matrix D and a vertex-topological feature matrix $X_0 \in R^{|V|xm}$, a multi-layer Graph Convolutional Network (GCN) ([10]) is a deep learning model, in which the layers are $X_{n+1} = \sigma(A \cdot X_n \cdot W_n)$, where W_n and $\sigma(\cdot)$ are as before. The output determines the predicted label for each vertex.

3 General Approaches

We propose the following approach for embedding large static graphs: choose an initial smaller group of vertices (e.g. the ones with the highest degree), calculate its embedding with any known embedding method (e.g. node2vec [6]) and use this embedding to calculate the whole graph embedding, where the new vertices are inserted one at a time. The calculation of the other vertices embedding would rely on the "basic" embedding approach used. Moreover, the method should have a short running time and a low memory usage.
After calculating the initial embedding, we adopt three approaches:

- **Weighted Combination:** In order to calculate a new vertex embedding, we use a weighted combination of its first and second neighbours' embeddings. This approach does not take into consideration the direction of the edges.

- **Regression:** We use regression to find the optimal parameters used to calculate the graph embedding from this initial embedding. Each individual graph has its own parameters with values that depend on its initial embedding. We have two kinds of calculations, both correspond to first and second order proximity (will be explained in details in the next section). This approach assumes the graph is directed and takes into consideration the direction of the edges.

- **Loss:** We use locally loss that will be minimized at each time step in order to find the optimal embedding of each vertex. This loss is based on the loss suggested in Graph Factorization algorithm [1] and corresponds to first order proximity with an added regularization.

4 Methods

In this section we will present specifically the problem and our four methods to solve it: The first corresponds to a weighted combination approach (OGRE), the second and third to a regression approach (DOGRE/WOGRE) and the last to a loss approach (LGF).Currently, the vertices of the initial embedding are chosen to be a fixed number of vertices with highest degree or highest k-core score (the maximal set of vertices in the graph such that the degree of each of them is at least k). The importance and influence of the size of the initial embedding and how the vertices need to be chosen, will be discussed later. Moreover, the initial embedding can be calculated with any static embedding algorithm. In our experiments we have used node2vec [6], Graph Factorization [1], HOPE [13] and GCN [10]. It is important to mention that all methods require a graph with minimal number of 10,000 vertices.

4.1 The Problem:

Given a graph $G = (V, E, W)$, a group of vertices H such that $H \subseteq V$ and an embedding of H - H^x, such that $H^x : i \to \overline{x}_i$ where $i \in H$ and $\overline{x}_i$ is its vector embedding, the goal is to find for every vertex $j \in G$ such that j is not in H, its embedding $\overline{x}_j$.

4.2 Online Graph Embedding (OGRE)

OGRE (**O**nline **GR**aph **E**mbedding) corresponds to a weighted combination approach which assumes an undirected graph.

Let H^x be the embedding of a chosen group of vertices $H \subseteq V$, composed of vectors embedding $\overline{x}_i$ for each vertex $i \in H$

For every vertex $j \in V$ two sets are defined:

1. The set of first order neighbours of vertex $j \in V$ that are in the embedding:

$$N_{1j} = \{n : (n, j) \in E \ , \ n \in H\} \quad , \quad |N_{1j}| = k_1$$

2. The set of second order neighbours of vertex $j \in V$ that are in the embedding:

$$N_{2j} = \{n : (n, a) \in E \ , \ (a, j) \in E, \ n, a \in H\} \quad , \quad |N_{1j}| = k_2$$

With these definitions, two average embeddings can be calculated:

1. The average embedding of the first order neighbours of vertex $j \in V$ that are in the embedding:

$$\overline{y}_1 = \frac{1}{k_1} \sum_{i=1}^{k_1} \overline{x}_{1i}$$

where x_{1i} is the embedding of vertex i such that $i \in N_{1j}$.

2. The average embedding of the second order neighbours of vertex $j \in V$ that are in the embedding:

$$\overline{y}_2 = \frac{1}{k_2} \sum_{i=1}^{k_2} \overline{x}_{2i}$$

where x_{2i} is the embedding of vertex i such that $i \in N_{2j}$.

Then the embedding of a vertex $j \in V$ is calculated by:

$$\overline{x}_j = \overline{y}_1 + \epsilon \cdot \frac{k_2}{k_1} \cdot (\overline{y}_1 - \overline{y}_2)$$

where ϵ is a small parameter (0.1, 0.01) representing the weight that is given to the influence of the second order neighbours.

Then, we get that $j \in H$ and $\overline{x}_j \in H^x$.

It is important to mention that the insertion order of the new vertices that are currently not in the embedding is depended on the size of N_{1j}, i.e. the first new vertex that is inserted to the embedding is the one with the highest number of first order neighbours that are already in the embedding. After every calculation, we update N_{1j} for each $j \in V$ and re-order the insertion order.

This is all done in an iterative manner until the only vertices that are left without an embedding are the ones for them $|N_{1j}| = 0$, i.e. they are not in the central connected component of the given graph. Therefore, after the online insertion process defined above is finished, a sub graph of these vertices is taken to be embed with the initial state-of-the-art embedding algorithm that was used to calculate the initial core. Then, these vertices also have an embedding.

At the end, we get that $H = V$, i.e. all graph vertices have a corresponding embedding vector in H^x, and the whole graph is embed.

4.3 Directed (Weighted) Online Graph Embedding (DOGRE / WOGRE)

DOGRE (**D**irected **O**nline **GR**aph **E**mbedding) and WOGRE (**W**eighted **O**nline **GR**aph **E**mbedding) correspond to a regression approach which assumes a directed graph (if an undirected graph is given, it is transformed to its directed version). Each one has its own formula to calculate a new vertex embedding. We will first mention necessary definitions common to both methods and then expand on each one separately.

Given a graph and an initial core of the graph that its embedding was already calculated with one of the state-of-the-art embedding algorithms, our goal is to find the other vertices embeddings, i.e. the whole graph embedding.

Let H^x be the embedding of a chosen group of vertices $H \subseteq V$, composed of vectors embedding $\overline{x}_i$ for each vertex $i \in H$

For every vertex $j \in V$ two sets are defined:

1. The set of first order incoming neighbours to vertex $j \in V$ that are in the embedding:
$$N_{1,j} = \{\, i \in V \mid (i,j) \in E\} \quad , \quad |N_{1,j}| = k_1$$
2. The set of first order outgoing neighbours from vertex $j \in V$ that are in the embedding:
$$N_{2,j} = \{\, i \in V \mid (j,i) \in E\} \quad , \quad |N_{2,j}| = k_2$$

Note that because the graph is directed, the direction of the edge matters, i.e. edge (i,j) is not the same edge as (j,i). That is why these both defenitions are necessary.

With these definitions, two average embeddings can be calculated:

1. The average embedding of the first order vertices incoming to j:
$$\overline{y}_{j,1} = \frac{1}{k_1} \sum \overline{x}_i \ , \ i \in N_{1,j}$$
2. The average embedding of the second order vertices outgoing from j:
$$\overline{y}_{j,2} = \frac{1}{k_2} \sum \overline{x}_i \ , \ i \in N_{2,j}$$

Then we can define for every $j \in V$: $z_{j,kl}$,where $z_{j,11}$ is the average embedding (i.e position) of vertices incoming to vertices incoming to j, $z_{j,12}$ is the average embedding of vertices outgoing from vertices incoming to j, $z_{j,21}$ is the average embedding of vertices incoming to vertices outgoing from j, $z_{j,22}$ is the average embedding of vertices outgoing from vertices outgoing from j.

From here, to calculate the embedding of vertex j, two options of regression are suggested.

4.3.1 Directed - DOGRE:

The embedding is defined as:

$$w_j = \alpha_1 \cdot y_{j,1} + \alpha_2 \cdot y_{j,2} - \sum \beta_{k,l} \cdot z_{j,kl}$$

where α_j , $\beta_{j,kl}$ are parameters with values which need to be extracted from the positions already in the graph (i.e from the position of vertices that are in the initial embedding) using linear regression.
The values of α and β determine the influence of $\overline{y}_{j,1}$, $\overline{y}_{j,2}$ and $z_{j,kl}$ on the final embedding of vertex j. Positive parameters correspond to a final embedding that is in the same direction as the given variable, while negative parameters correspond to a final embedding that is in the opposite direction. Big values mean more consideration in the variable and its direction, while smaller ones mean less consideration.

4.3.2 Weighted - WOGRE

Let us define k^j_{in} and k^j_{out} as the number of vertices incoming to and outgoing from vertex j, respectively. Similarly, $k^j_{in,in}$ is the number of vertices incoming to vertices incoming to j, $k^j_{in,out}$ is the number of vertices outgoing from vertices incoming to j, $k^j_{out,in}$ is the number of vertices incoming to vertices outgoing from j and $k^j_{out,out}$ is the number of vertices outgoing from vertices outgoing from j. Then:

$$\begin{cases} p_{j,1} = \alpha_1 \cdot y_{j,1} + \alpha_2 \cdot y_{j,1} \cdot k_{j,in} + \alpha_3 \cdot y_{j,1} \cdot (k^j_{in})^2 \\ p_{j,2} = \alpha_4 \cdot y_{j,2} + \alpha_5 \cdot y_{j,2} \cdot k_{j,out} + \alpha_6 \cdot y_{j,2} \cdot (k^j_{out})^2 \\ p_{j,11} = \alpha_7 \cdot z_{j,11} + \alpha_8 \cdot z_{j,11} \cdot k^j_{in,in} + \alpha_9 \cdot z_{j,11} \cdot (k^j_{in,in})^2 \\ p_{j,12} = \alpha_{10} \cdot z_{j,12} + \alpha_{11} \cdot z_{j,12} \cdot k^j_{in,out} + \alpha_{12} \cdot z_{j,12} \cdot (k^j_{in,out})^2 \\ p_{j,21} = \alpha_{13} \cdot z_{j,21} + \alpha_{14} \cdot z_{j,21} \cdot k^j_{out,in} + \alpha_{15} \cdot z_{j,21} \cdot (k^j_{out,in})^2 \\ p_{j,22} = \alpha_{16} \cdot z_{j,22} + \alpha_{17} \cdot z_{j,22} \cdot k^j_{out,out} + \alpha_{18} \cdot z_{j,22} \cdot (k^j_{out,out})^2 \end{cases}$$

$$\Rightarrow w_j = p_{j,1} + p_{j,2} + p_{j,11} + p_{j,12} + p_{j,21} + p_{j,22}$$

where α_l , $l \in [1, 18]$, are parameters with values which need to be extracted from the positions already in the graph (i.e from the position of vertices that are in the initial embedding) using linear regression.
The values of α_l determine the impact of their corresponding average embeddings. Furthermore, each p is a polynomial sum, therefore $alpha_l$ also multiplied by the corresponding number of vertices in the sets defined above, with a changed power.

4.4 Local Graph Factorization (LGF)

LGF (**L**ocal **G**raph **F**actorization) corresponds to a loss approach which assumes the graph is directed. The loss of Graph Factorization [1] is being used here, therefore the

initial embedding for this method is done with Graph Factorization algorithm only.

The loss function of Graph Factorization algorithm is:

$$L = \frac{1}{2} \sum_{(i,j)\in E} (Y_{ij} - < Z_i, Z_j >)^2 + \frac{\lambda}{2} \sum_i ||Z_i||^2$$

where Y_{ij} is the weight of the edge (i, j), Z is the factor matrix with embedding vector Z_i for vertex i and λ is a regularization parameter.

The goal is to find the embedding of vertex i that is still not in the embedding, therefore in our calculations we assume all embeddings (positions) z_j are fixed, except z_i which is the embedding of vertex i. As a result, the loss function can be written as:

$$L = \frac{1}{2} \sum_{j\in N(i)} (Y_{ij} - z_i^T z_j)^2 + \frac{\lambda}{2} ||Z_i||^2$$

This new loss function should be minimized by z_i:

$$\frac{\partial L}{\partial z_i} = 0$$

$$\frac{\partial L}{\partial z_i} = \sum_{j\in N(i)} [(Y_{ij} - z_i^T z_j)(-z_j)] + \lambda z_i = 0$$

$$\lambda z_i = \sum_{j\in N(i)} [(Y_{ij} - z_i^T z_j) z_j] = 0$$

$$\lambda z_i = \sum_{j\in N(i)} Y_{ij} z_j - \sum_{j\in N(i)} (z_i^T z_j) z_j = 0$$

A system of linear equations that needs to be solved is gotten:

$$z_i^l = \frac{1}{\lambda} [\sum_{j\in N(i)} Y_{ij} z_j^l - \sum_{j\in N(i)} \sum_k (z_i^k z_j^k) z_j^l] = 0$$

$$z_i^l = \frac{1}{\lambda} [\sum_{j\in N(i)} Y_{ij} z_j^l - \sum_k \sum_{j\in N(i)} (z_j^k z_j^l) z_i^k] = 0$$

To make this equation even easier, matrices will be used instead. One can define:

$$c_i^l = \sum_{j\in N(i)} Y_{ij} z_j^l$$

$$b_i^k = \sum_{j\in N(i)} (z_j^k z_j^l)$$

And get:

$$\lambda z_i^l = c_i^l - \sum_k b_i^k z_i^k = 0$$

$$\lambda Z = C - BZ$$

$$\lambda IZ + BZ = C$$

$$(\lambda I + B)Z = C$$

$$Z = (\lambda I + B)^{-1} C$$

And this is the final embedding of vertex i where $Z \in R^{dx1}$, $B \in R^{dxd}$ and $C \in R^{dx1}$.

5 Evaluation Tasks

Embeddings are graph representations, thus they can be used in a variety of tasks. In order to evaluate the performance of our proposed methods and measure their quality, we suggest three tasks that can be expand in the future.

5.1 Node Classification

Predicting labels using network topology is widely popular in network analysis and has variety of applications, including document classification and interest prediction. A good network embedding should capture the network structure and hence be useful for node classification. As done in [5], we evaluate and compare the effectiveness of our suggested methods and state-of-the-art algorithms on this task by using the generated embedding as vertex features to classify the vertices. The vertex features are input to a one-vs-rest logistic regression. Then we commit two types of tasks:

1. We compare between state-of-the-art embedding methods and our proposed one. For each data set, we randomly sample 10% to 90% of vertices as training data and evaluate the performance on the remaining vertices. We perform this split 10 times and report the mean with confidence interval. We evaluate the performance with three scores: micro-F1, macro-F1 and AUC.

2. Only for our suggested method, we check the influence of the number of vertices in the initial embedding on the success of the node classification mission, i.e. we plot graphs of each score as a function of number of vertices in the initial embedding.

5.2 Link Prediction

Networks are constructed from the observed iterations between entities, which may be incomplete or inaccurate. The challenge often lies in identifying spurious interactions and predicting missing information. Link prediction refers to the task of predicting either missing interactions or links that may appear in the future in an evolving network.

Embeddings capture inherent dynamics of the network either explicitly or implicitly thus enabling application to link prediction.
We commit the link prediction task both on our methods and state-of-the-art methods in order to compare between them. Given an edge and its to ends x and y, we define the edge's representation as $|e_x - e_y|$, where e_x, e_y are x's y's embeddings respectively, and $|\cdot|$ indicates norm in order one between two vectors. This definition works no matter if two vertices have an edge between them or not, therefore can be utilized as features for link prediction. We randomly hold out a fixed number of edges, say K, as positive edges, and also choose K false ones (i.e. two vertices have no edge). We then train logistic regression classifier, where the features are the norm of each edge and the labels are 0 for true edge and 1 for false one. We use Micro-F1, Macro-F1 and AUC as our success measurements.
As explained above in node classification task, we here too compare ourselves to state-of-the-art embedding algorithms, as well as checking the influence of the number of nodes in the initial embedding on the success of link prediction task.

5.3 Visualization

Since embedding is a low-dimensional vector representation of vertices in the graph, it allows us to visualize the vertices to understand the network topology. As different embedding methods preserve different structures in the network, their ability and interpretation of node visualization differ. For instance, embeddings learned by node2vec with parameters set to prefer BFS random walk would cluster structurally equivalent vertices together. As done in [5], we compare the ability of state-of-the-art embedding methods to our proposed method to visualize vertices on a few data-sets. We learn a 10-dimensional embedding for each method and input it to t-SNE to visualize the vertices in a 2-dimensional space. For labeled graphs we also color the vertices by their labels.

6 Experimental Setup

Our experiments evaluate the feature representations obtained using our methods and state-of-the-art methods, both reviewed before in the previous sections. We here specify the datasets and evaluation metrics used.

6.1 Datasets

We evaluate the embedding approaches on 5 real datasets. The datasets are summarized in Table 1 .

- **DBLP, Cora, Pubmed:** These are citation networks in which vertices denote papers and edges represent citation relations.
- **Reddit:** A post-post graph. A vertex represents a post. An edge exists between two posts if the same user has commented on both posts

- **Yelp:** A social network graph. A vertex is a user. An edge represents friendship. Vertex attributes are user comments converted from text using Word2Vec.

Dataset	$\|V\|$	$\|E\|$	Number of Labels
DBLP	17,716	105,734	4
Pubmed	19,917	79,612	3
Cora	19,793	65,311	70
Reddit	232,965	11,606,919	41
Yelp	716,847	6,977,410	100

Table 1: Datasets Statistics

6.2 Evaluation Metrics

To evaluate the performance of embedding methods on link prediction, we use AUC as our metric. For node classification, we use Micro-F1 and Macro-F1. These metrics are defined as follows:

- **AUC**: An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters: True Positive Rate and False Positive Rate. AUC stands for "Area under the ROC Curve." That is, AUC measures the entire two-dimensional area underneath the entire ROC curve. AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example. AUC ranges in value from 0 to 1. A model whose predictions are 100% wrong has an AUC of 0.0; one whose predictions are 100% correct has an AUC of 1.0.

- **Macro-F1**, in a multi-label classification task, is defined as the average $F1$ of all the labels, i.e.,
$$Macro - F1 = \frac{\sum_{l \in L} F1(l)}{|L|}$$
where $F1(l)$ is the F1-score for label l .

- **Micro-F1** calculates $F1$ globally by counting the total true positives, false negatives and false positives, giving equal weight to each instance. It is defined as follows:
$$Micro - F1 = \frac{2 \cdot P \cdot R}{P + R}$$
where $P = \frac{\sum_{l \in L} TP(l)}{\sum_{l \in L}(TP(l)+FP(l))}$ and $R = \frac{\sum_{l \in L} TP(l)}{\sum_{l \in L}(TP(l)+FN(l))}$, are precision (P) and recall (R) respectively, and $TP(l)$, $FP(l)$ and $FN(l)$ denote the number of true positives, false positives and false negatives respectively among the instances which are associated with the label l either in the ground truth or the predictions.

7 Experiments and Analysis

In this section, we evaluate and compare embedding methods on the three tasks presented above- link prediction, node classification and visualization, as well as discuss the results and show important graphs. Furthermore, we compare the running time and memory usage of all presented methods (ours and state-of-the-art). Note that the results for GCN as an initial embedding and comparison to state-of-the-art GCN apears in a seperate section, as well as the discussion.

7.1 Rational

Our embedding methods begin with an initial embedding, i.e. vertices with highest degree are chosen to be embed with a known embedding algorithm such as node2vec, HOPE and Graph Factorization. Specifically, for node classification task, OGRE with an small initial seed of 100 nodes obtains an accuracy higher than all other algorithms, both state-of-the-art and OGRE with larger size of initial embedding. In order to find a logical explanation to this, we have checked the node classification performance individually on the vertices of the core, and individually on the rest of the vertices (periphery). Specifically, we divided the vertices into two groups: core vertices (the vertices that are in the initial embedding) and periphery vertices (the rest of the vertices). Instead of performing the task on all vertices, the task (as explained in the fifth section) is done two times, each time for a different group of vertices. Test ratio is fixed to 0.2 (i.e. 80% of the vertices are for the train set and 20% are for the test set). Figure 5 presents Micro-F1 and Macro-F1 scores on node classification as a function of initial embedding size on Pubmed dataset, both for core and periphery vertices. Each line represents a different type of group of vertices ("initial" is the core vertices and "per" is the periphery vertices) and the graph was embed with node2vec embedding algorithm. It can be easily seen that the performance of the core is much better than the periphery performance by approximately 50%, i.e. node classification task works best for the core, specifically for the smallest one, and as the size of the core increases, the performance decreases. As for the periphery, for all initial sizes the performance is poor. An obvious conclusion is that there is no point in calculating the embedding with node2vec for more than 100 vertices - it is better to take the most important ones (e.g. with highest degree) and embed only them with node2vec. Therefore a rational explanation for OGRE success is that the embedding of the core works best for node classification task in contrast to the periphery performance which is poor, so there is no justification to embed the whole graph.

7.2 Results

Link Prediction and Node Classification tasks are done as explained in the fifth section. For link prediction task, we choose 10000 to be the number of true and false edges. Tables 2 and 3 presents the results for all datasets for each method for link prediction and node classification tasks, respectively, where test ratio is fixed to 0.2. For our methods, the

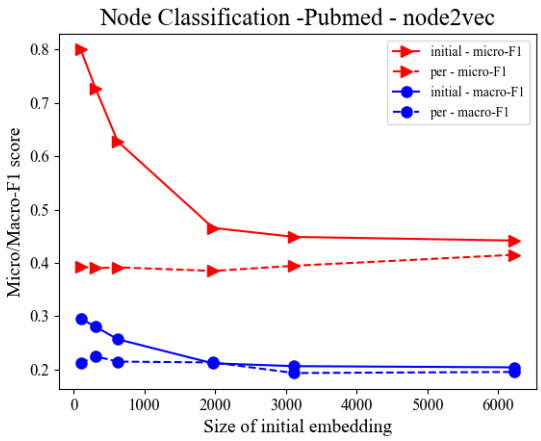


Figure 1: Node2vec performance on node classification task

graph is embed with different sizes of initial embedding, and the best result is chosen to appear in the table. Also, the initial vertices were chosen by their degree (and not k-core score) because it has yielded better results on both tasks (wil be discussed later). Note that "node2vec + OGRE" embedding method corresponds to OGRE method with an initial embedding calculated with node2vec algorithm. The rest of the methods are defined in the same logic.

7.2.1 Link Prediction Performance

AUC Score On Link Prediction Task					
Method	**DBLP**	**Pubmed**	**Cora**	**Reddit**	**Yelp**
node2vec	0.496	0.517	0.563	✗	✗
HOPE	**0.645**	0.688	**0.598**	✗	✗
GF	0.517	**0.705**	0.564	✗	✗
node2vec + OGRE	0.691	0.811	**0.812**	**0.714**	**0.818**
HOPE + OGRE	**0.807**	0.803	0.806	0.713	0.790
GF + OGRE	0.750	**0.815**	0.775	0.680	0.787
node2vec + DOGRE	0.592	0.686	0.564	0.568	0.617
HOPE + DOGRE	0.608	0.628	0.582	0.530	0.533
GF + DOGRE	0.606	0.624	0.593	0.593	0.725
node2vec + WOGRE	0.568	0.648	0.628	✗	✗
HOPE + WOGRE	0.630	0.639	0.596	✗	✗
GF + WOGRE	0.610	0.610	0.593	✗	✗
GF + LGF	0.537	0.552	0.526	✗	✗

Table 2: Link Prediction Performance, test ratio is fixed to 0.2

7.2.2 Node Classification Performance

Micro-F1 and Macro-F1 Scores On Node Classification Task						
Method	**Metric**	**DBLP**	**Pubmed**	**Cora**	**Reddit**	**Yelp**
node2vec	Macro-F1	0.296	0.197	0.164	✗	✗
	Micro-F1	0.497	0.4	**0.233**		
HOPE	Macro-F1	0.165	0.229	0.02	✗	✗
	Micro-F1	0.451	0.396	0.078		
GF	Macro-F1	0.245	0.289	**0.186**	✗	✗
	Micro-F1	0.461	0.395	0.217		
node2vec + OGRE	Macro-F1	0.433	0.414	0.008	**0.665**	✗
	Micro-F1	0.576	0.555	0.055	**0.784**	
HOPE + OGRE	Macro-F1	**0.617**	**0.692**	0.036	0.404	✗
	Micro-F1	**0.713**	**0.730**	0.172	0.632	
GF + OGRE	Macro-F1	0.523	0.629	0.049	0.289	✗
	Micro-F1	0.65	0.668	0.177	0.530	
node2vec + DOGRE	Macro-F1	0.31	0.235	0.008	0.129	✗
	Micro-F1	0.537	0.401	0.067	**0**.312	
HOPE + DOGRE	Macro-F1	0.165	0.238	0.005	0.011	✗
	Micro-F1	0.481	0.402	0.056	0.114	
GF + DOGRE	Macro-F1	0.205	0.282	0.01	0.025	✗
	Micro-F1	0.474	0.402	0.06	0.14	
node2vec + WOGRE	Macro-F1	0.280	0.258	0.016	✗	✗
	Micro-F1	0.521	0.4	0.089		
HOPE + WOGRE	Macro-F1	0.166	0.284	0.089	✗	✗
	Micro-F1	0.480	0.4	0.198		
GF + WOGRE	Macro-F1	0.215	0.299	0.009	✗	✗
	Micro-F1	0.482	0.404	0.06		
GF + LGF	Macro-F1	0.177	0.273	0.014	✗	✗
	Micro-F1	0.476	0.398	0.061		

Table 3: Node Classification Performance, test ratio is fixed to 0.2

7.2.3 Conclusions

- For all smaller datasets (DBLP, Cora and Pubmed) at least one of our suggested methods has outperformed all state-of-the-art algorithms that were used, and by high percents.
- For all datasets, it seems that OGRE has performed much better than DOGRE, WOGRE and LGF.
- LGF is both slow and has a much worse performance than the others on the smaller

datasets, therefore it has not been used for the large datasets (Reddit and Yelp). Similarly, from the same reasons WOGRE is not used for the large datasets.

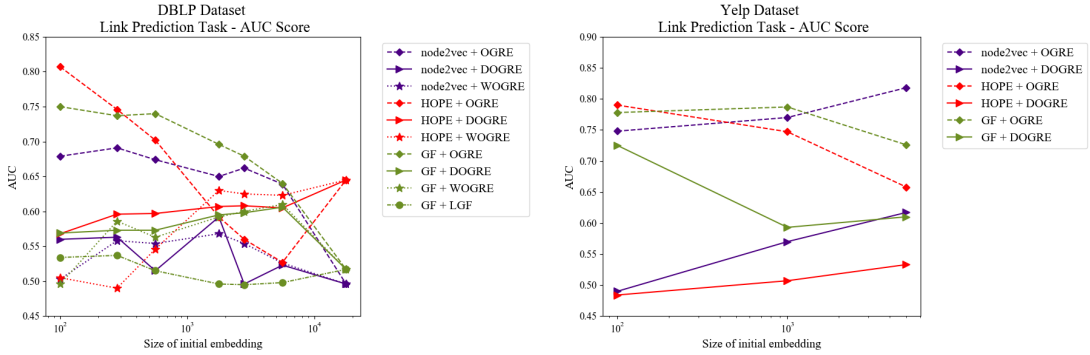
- As mentioned in the beginning, as opossed to their claims, the state-of-the-art algorithms- node2vec, Graph Factorization and HOPE - have all failed to work on the large and dense datasets (Yelp and Reddit). Therefore no results are presented for them.

- From the results above it seems that HOPE is the best state-of-the-art embedding method for both link prediction and node classification, therefore OGRE with an initial embedding of HOPE outperformed all other state-of-the-art algorithms (including HOPE) and all other combinations (our methods) in almost all datasets. This combination works best because the important relations and connections are captured well in HOPE on small data (the initial embedding) and then expanding it with OGRE keeps capturing important information.

7.3 Influence of Initial Embedding Size

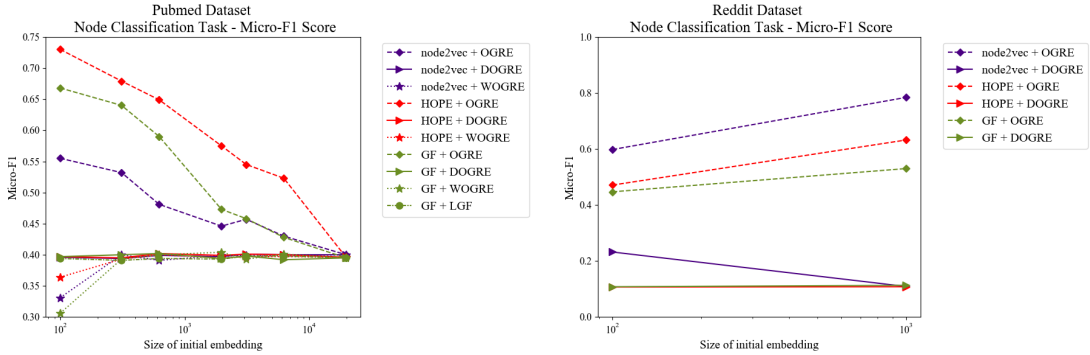
Figure 2 presents the influence of the initial embedding size on the performance in link prediction task on AUC score for DBLP and Yelp datasets. Figure 3-4 present the influence of the initial embedding size on the performance in node classification task, on Micro-F1 and Macro-F1 respectively, for Pubmed and Reddit datasets. The last initial size is actually the size of the graph, i.e. the last point represents the performance of the state-of-the art method which was used.

Looking on figures 1-3, one can notice that for relatively small graphs (DBLP/Pubmed) OGRE method has given the best results and for the smallest size of initial embedding (=100). A logical explanation is that because the graph is small and not dense enough, a small initial embedding that does not take into consideration the direction of the edges is enough to capture all necessary information. A more complicated embedding that does take into account the direction of first and second order neighbours of each vertex and uses regression is just too complicated and unnecessary. However, for large dense graphs such as Yelp and Reddit, a bigger size of initial embedding actually yields better results, until a size limitation - it can be seen that when the size is bigger than 10^4, the performance is worse. This may happen because although the graph is large and dense and therefore the size of initial embedding needs to be larger as well, it cannot be that large because then we might miss important connections through OGRE. One needs to remember that the initial core is embed with state-of-the-art method that might not be so good to embed with, thus a limitation on the initial size is needed.
For DOGRE and WOGRE methods, in DBLP/Pubmed, the performance approximately increases with the size of initial embedding. For large graphs such as Yelp and Reddit, it is approximately the same. In here too we have a limitation for the initial embedding size.



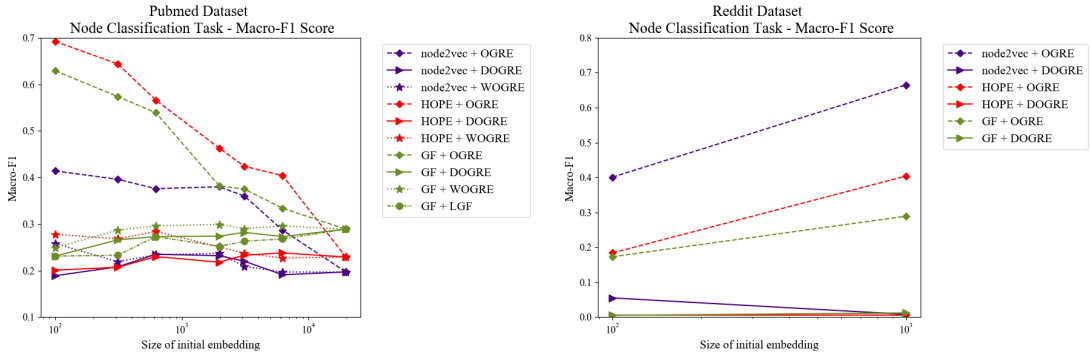
(a) DBLP (b) Yelp

Figure 2: Link prediction task- initial embedding size affect on AUC score



(a) Pubmed (b) Reddit

Figure 3: node classification task- initial embedding size affect on Micro-F1 score



(a) Pubmed (b) Reddit

Figure 4: node classification task- initial embedding size affect on Macro-F1 score

7.4 Core VS Periphery

Our embedding methods begin with an initial embedding, i.e. vertices with highest degree are chosen to be embed with a state-of-the-art embedding algorithm. Looking

on figures 2-4, we wanted more than a logical explanation to why for OGRE method a small size of initial embedding works best. In order to do that, we have checked the link prediction and node classification performance specifically on the vertices of the core, and specifically on the rest of the vertices (periphery). The performance is measured for both state-of-the-art algorithms and our methods.

7.4.1 Link Prediction

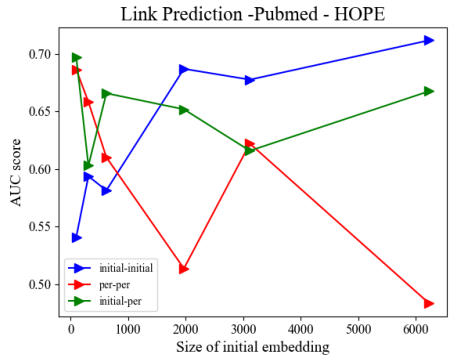
For link prediction task, we divided the edges into three groups: edges occurring between two core vertices (initial-initial), edges occurring between two periphery vertices (per-per) and edges occurring between a core vertex to a periphery vertex (initial-per). Instead of performing the task on all edges, the task (as explained in the fifth section) is done three times, each time for a different group of edges. Test ratio is fixed to 0.2 (i.e. 80% of the edges are for the train set and 20% are for the test set) and the number of true and false edges is 10000 (if one of the groups has less than 10000 edges, then all edges consider as positive ones and an equal number of negative edges are chosen.
Figure 5 presents AUC score on link prediction as a function of initial embedding size on Pubmed dataset, both for core and periphery vertices. Each line represents a different type of group. The graph was embed with OGRE with HOPE as the initial embedding and the comparison is to HOPE.

7.4.2 Node Classification

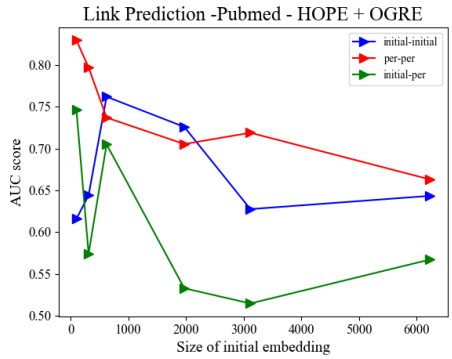
For node classification task, we divided the vertices into two groups: core vertices (the vertices that are in the initial embedding) and periphery vertices (the rest of the vertices). Instead of performing the task on all vertices, the task (as explained in the fifth section) is done two times, each time for a different group of vertices. Test ratio is fixed to 0.2 (i.e. 80% of the vertices are for the train set and 20% are for the test set).
Figure 6 presents Micro-F1 and Macro-F1 scores on node classification as a function of initial embedding size on Pubmed dataset, both for core and periphery vertices. Each line represents a different type of group of vertices. The graph was embed with OGRE with node2vec as the initial embedding and the comparison is to node2vec.

7.5 Tasks Performance as Function of Test Ratio

Figures 7-9 present the influence of test ratio on link prediction and node classification performance. Test ratio equals to 0.2 means 80% of the data is for the train set and 20% is for the test set. Note that for our methods, the graph is embed with different size of initial embedding, and then we choose the best result to appear in the graph. It can be seen that For both link prediction and node classification tasks it can be seen the the test ratio has almost no impact on the performance.

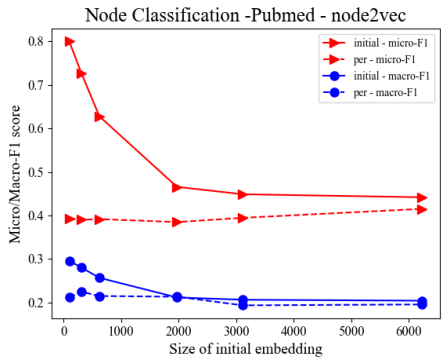


(a) HOPE

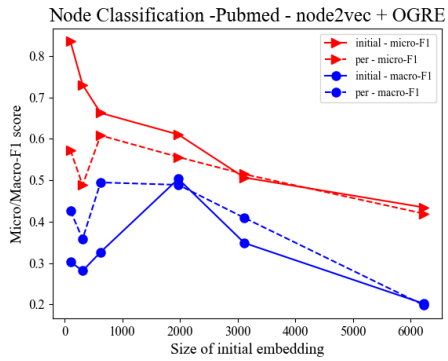


(b) HOPE + OGRE

Figure 5: Link prediction performance

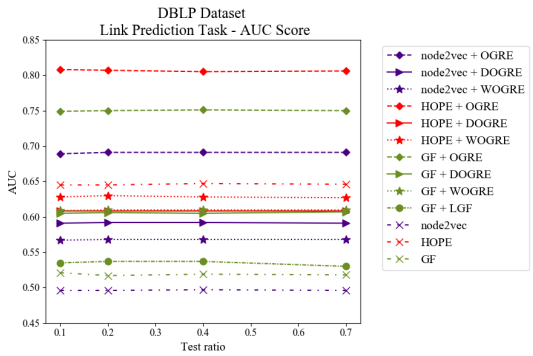


(a) node2vec

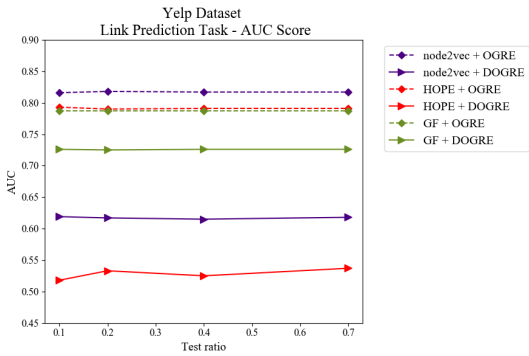


(b) node2vec + OGRE

Figure 6: Node classification performance

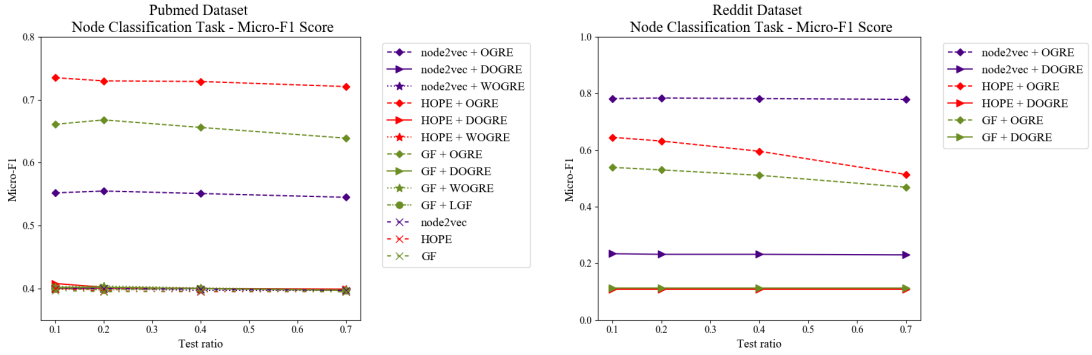


(a) DBLP



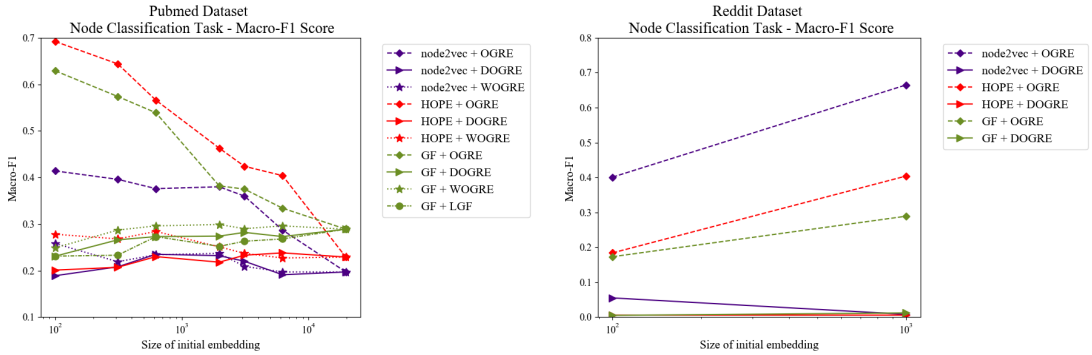
(b) Yelp

Figure 7: Best results for link prediction - AUC score



(a) Pubmed (b) Reddit

Figure 8: Best results for node classification - Micro-F1 score

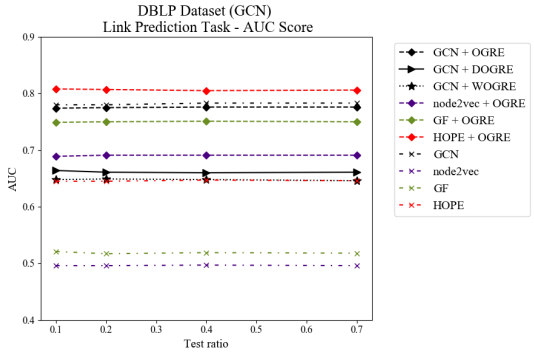


(a) Pubmed (b) Reddit

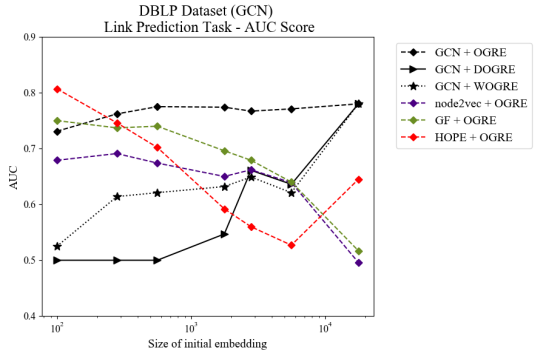
Figure 9: Best results for node classification - Macro-F1 score

7.6 GCN Core

In the previous sections, node2vec, HOPE and Graph Factorization embedding algorithm were compared to our suggested methods. Here we show the results for a combination of GCN, a state-of-the-art embedding algorithm for mainly node classification task, and OGRE which had the best performance in all tasks. Figures 10-11 present AUC scroe and Micro/Macro-F1 scores for link prediction and node classification, respectively.
It can be seen that the performance of node classification task significantly improving when the graph is embed with a combination of GCN and OGRE. Furthermore, because GCN is a powerful graph classifier, DOGRE and WOGRE that do not perform well when combined with node2vec/graph factorization/HOPE, perform very well when combined with GCN.

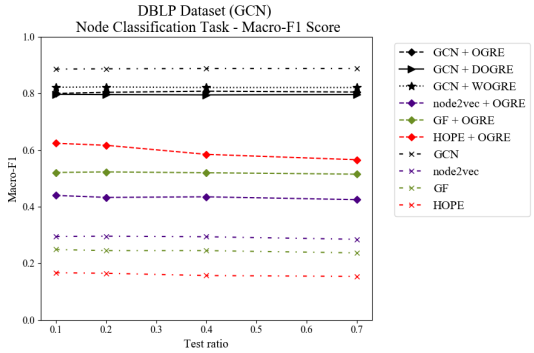


(a) Best Results of Link Prediction

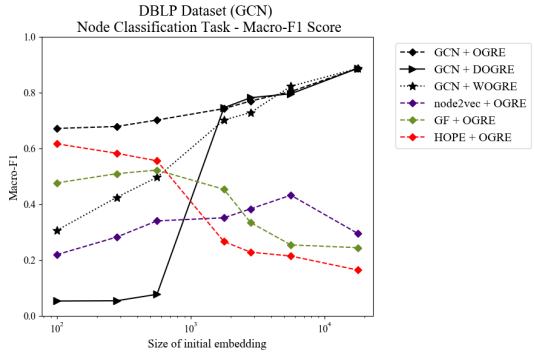


(b) Initial embedding size affect on AUC score

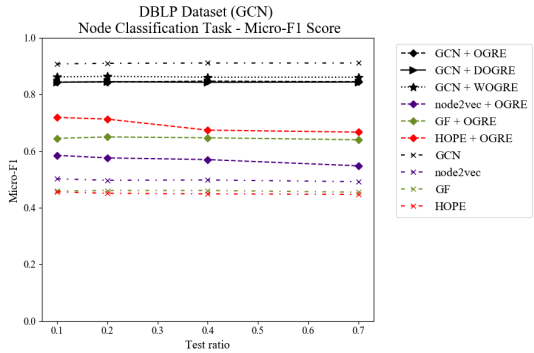
Figure 10: DBLP Link Prediction



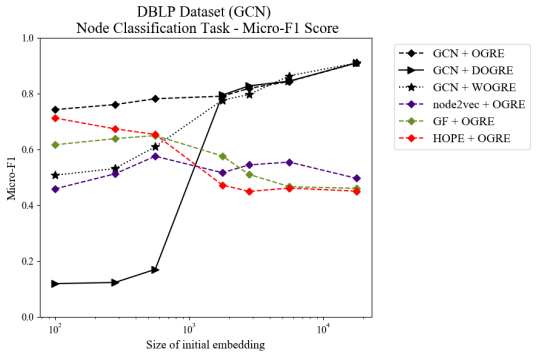
(a) Best Results of Node Classification Macro-F1 score



(b) Initial embedding size affect on Macro-F1 score



(c) Best Results of Node Classification Micro-F1 score



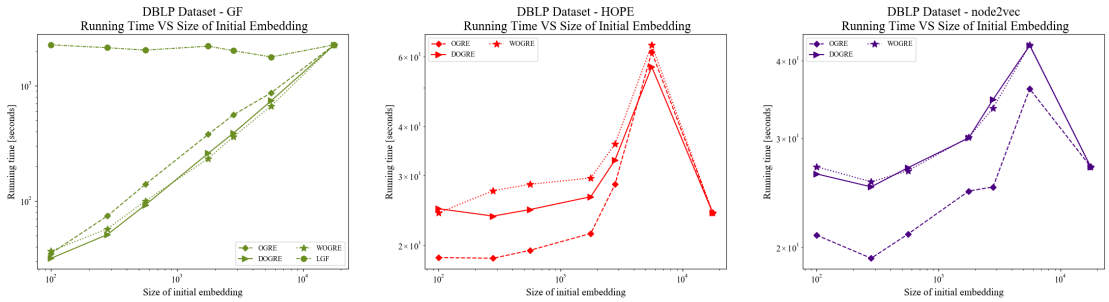
(d) Initial embedding size affect on Micro-F1 score

Figure 11: DBLP Node Classification

7.7 Computational cost of existing methods for large graphs

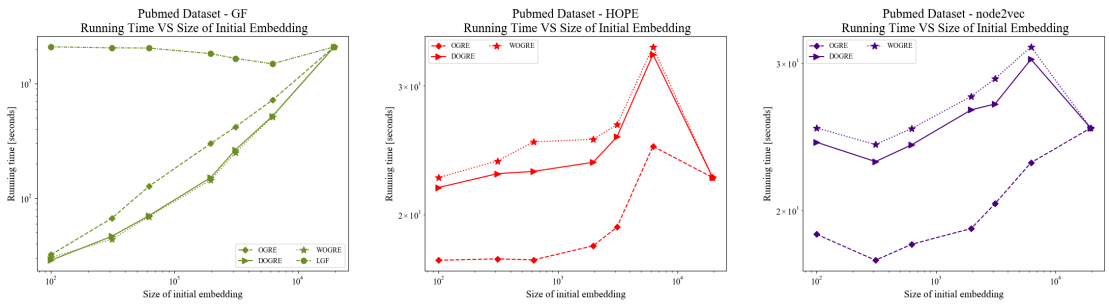
Figures 12-15 show the comparison of the computational cost of exiting methods and our suggested methods. The x-axis corresponds to the size of initial embedding, while

the y-axis corresponds to the running time of each method in seconds. The final size of initial embedding is equal to the number of vertices in the graph, i.e. the final dot in the graph represents the running time of the state-of-the-art algorithm that was used. The graphs are in log-log scale.
It can be seen that for relatively small and not dense graphs our methods are a little more expensive than existing methods, except for Graph Factorization algorithm. However, for large dense graphs, all three exiting methods failed to embed the graph (due to memory issues) while ours did work in a reasonable time. As expected, approximately, the running time is growing with the size of initial embedding for all different approaches, with all different state-of-the-art methods as initial embedding. Furthermore, for all graphs OGRE method is less expensive than the others and it is most noticeable in the large graphs, where the running time of OGRE is significantly shorter than the others.



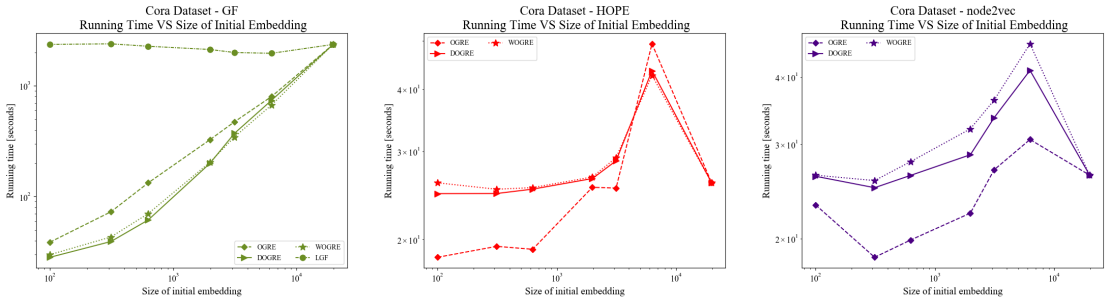
(a) Graph Factorization (b) HOPE (c) node2vec

Figure 12: DBLP Running Time



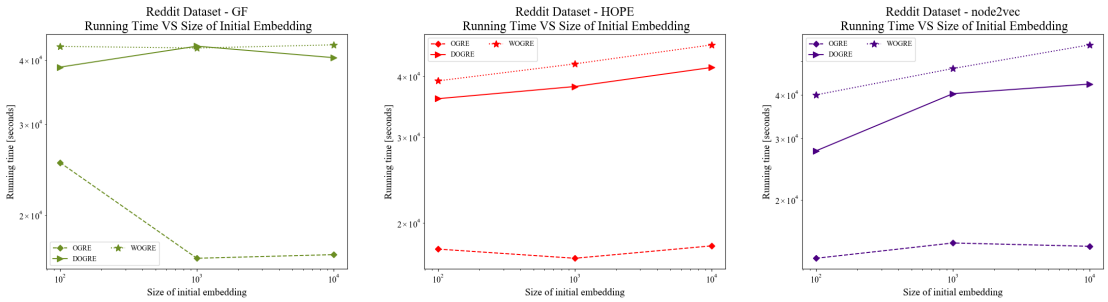
(a) Graph Factorization (b) HOPE (c) node2vec

Figure 13: Pubmed Running Time



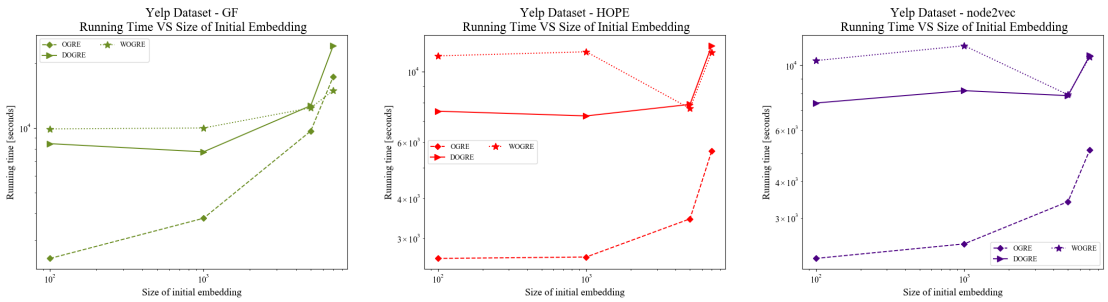
(a) Graph Factorization (b) HOPE (c) node2vec

Figure 14: Cora Running Time



(a) Graph Factorization (b) HOPE (c) node2vec

Figure 15: Reddit Running Time



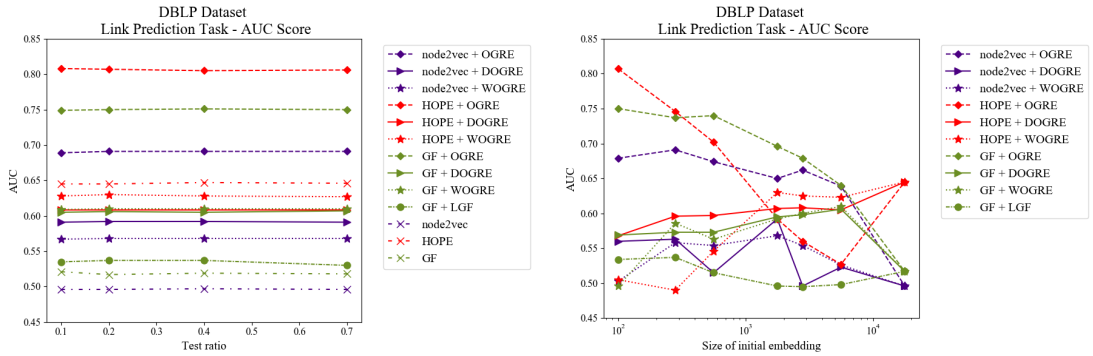
(a) Graph Factorization (b) HOPE (c) node2vec

Figure 16: Yelp Running Time

8 Appendix

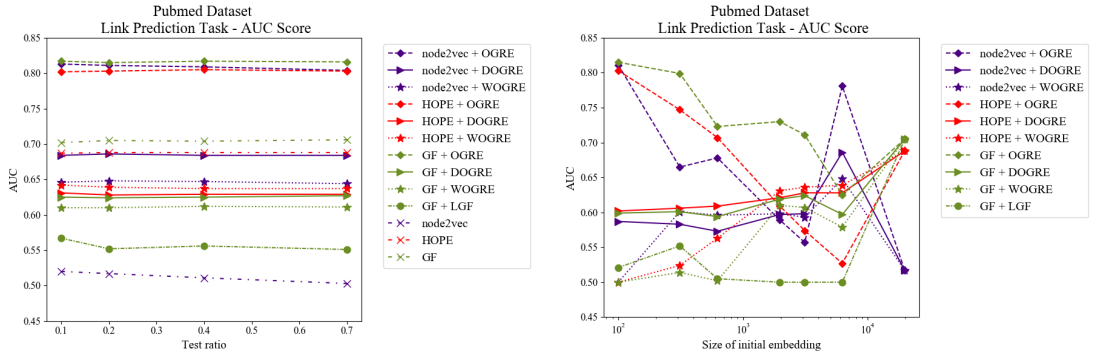
Appendix include figures of link prediction, figures of node classification and visualizations on DBLP dataset with different embedding methods and combination (visualized with TSNE). From visualization figures one can notice that partition by labels (visualized by the colors) is very obvious when embedding the graph with one of our methods with a small size of initial embedding. However, when embedding with known algorithms such as node2vec, the partition is pretty messy, supporting the results shown in the previous sections.

8.1 Figures Link Prediction



(a) Best Results of Link Prediction (b) Initial embedding size affect on AUC score

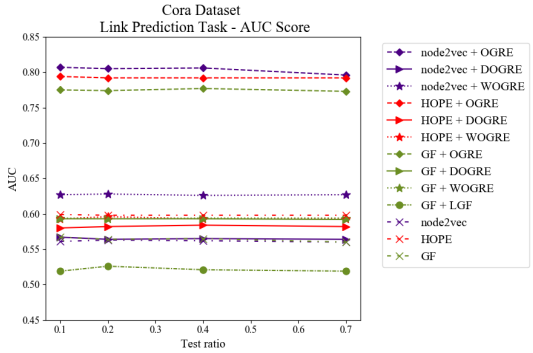
Figure 17: DBLP Link Prediction



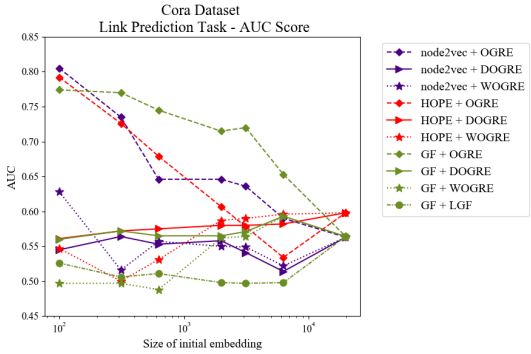
(a) Best Results of Link Prediction (b) Initial embedding size affect on AUC score

Figure 18: Pubmed Link Prediction

8.2 Figures Node Classification

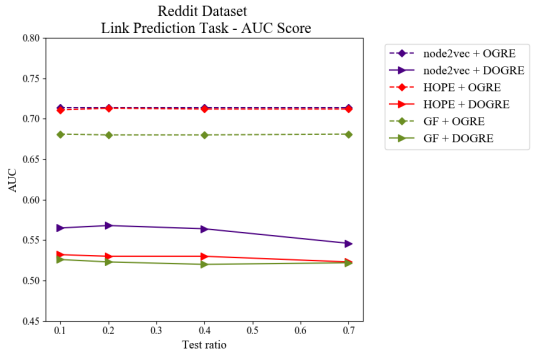


(a) Best Results of Link Prediction

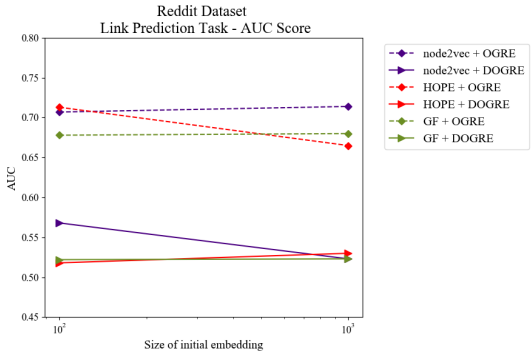


(b) Initial embedding size affect on AUC score

Figure 19: Cora Link Prediction

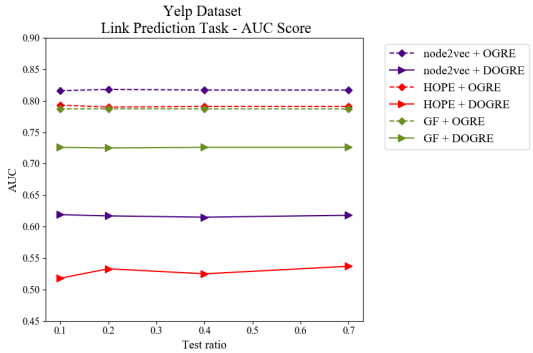


(a) Best Results of Link Prediction

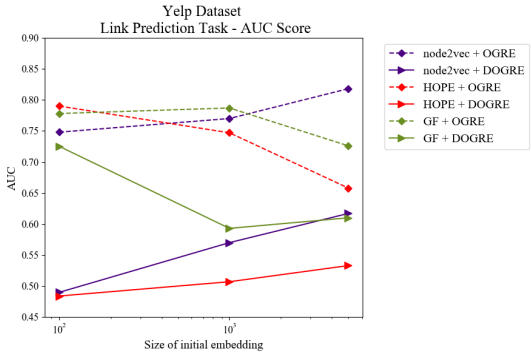


(b) Initial embedding size affect on AUC score

Figure 20: Reddit Link Prediction

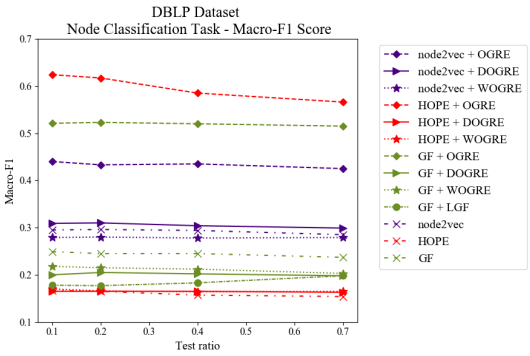


(a) Best Results of Link Prediction

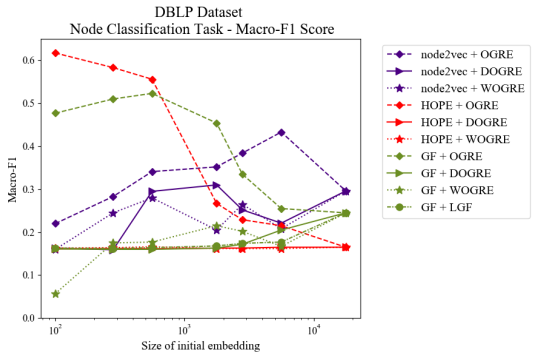


(b) Initial embedding size affect on AUC score

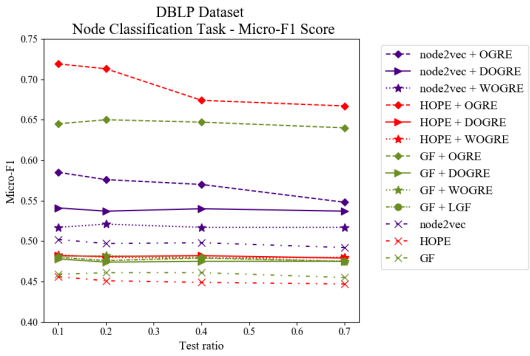
Figure 21: Yelp Link Prediction



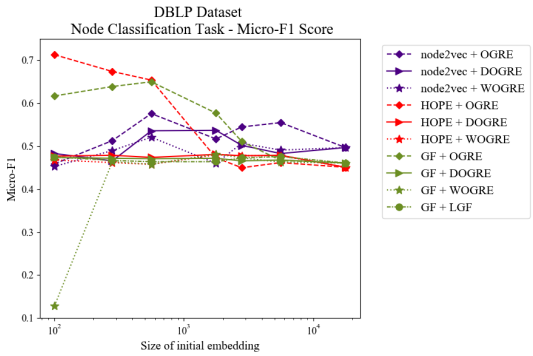
(a) Best Results of Node Classification Macro-F1 score



(b) Initial embedding size affect on Macro-F1 score

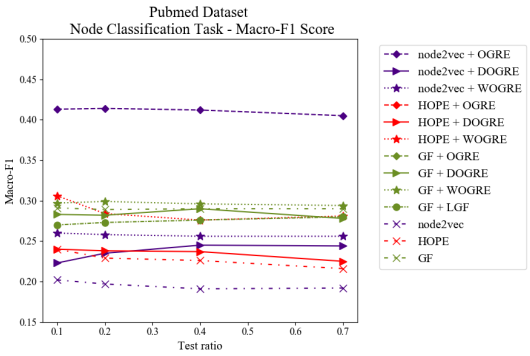


(c) Best Results of Node Classification Micro-F1 score

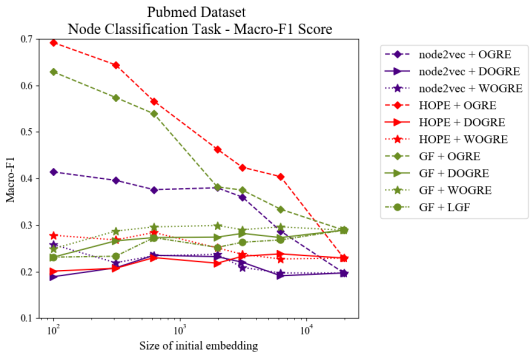


(d) Initial embedding size affect on Micro-F1 score

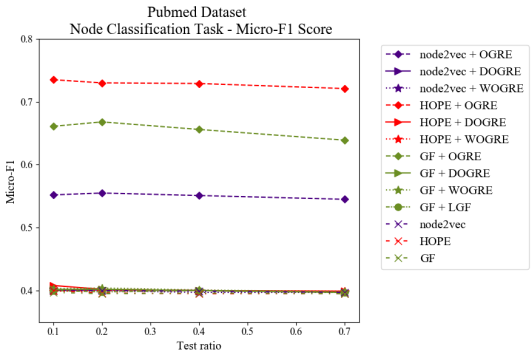
Figure 22: DBLP Node Classification



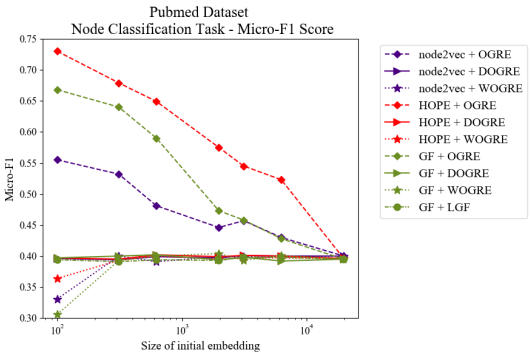
(a) Best Results of Node Classification Macro-F1 score



(b) Initial embedding size affect on Macro-F1 score

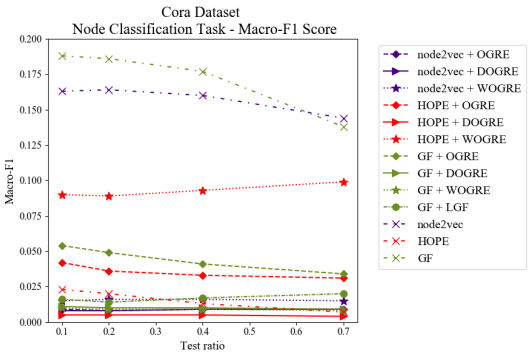


(c) Best Results of Node Classification Micro-F1 score

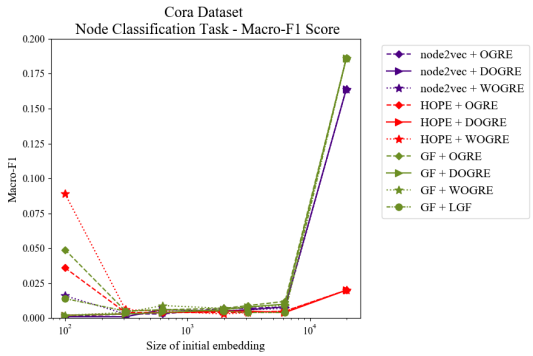


(d) Initial embedding size affect on Micro-F1 score

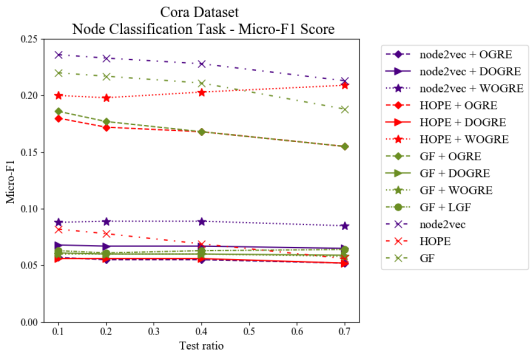
Figure 23: Pubmed Node Classification



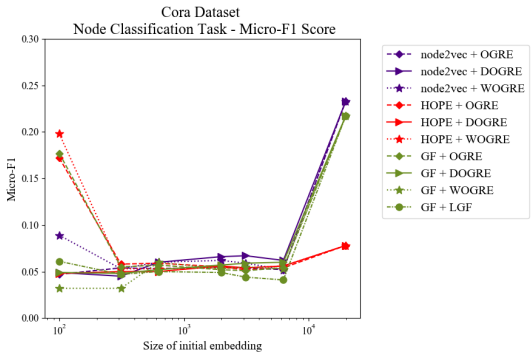
(a) Best Results of Node Classification Macro-F1 score



(b) Initial embedding size affect on Macro-F1 score

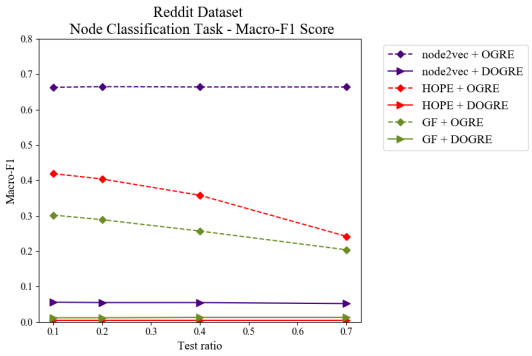


(c) Best Results of Node Classification Micro-F1 score

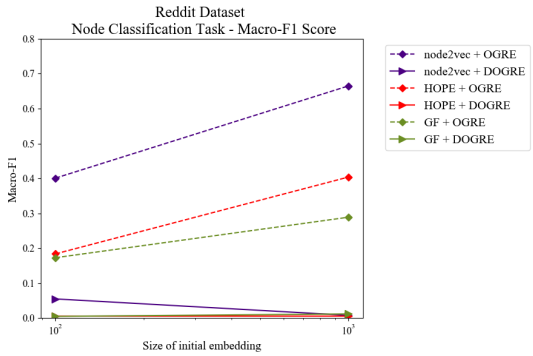


(d) Initial embedding size affect on Micro-F1 score

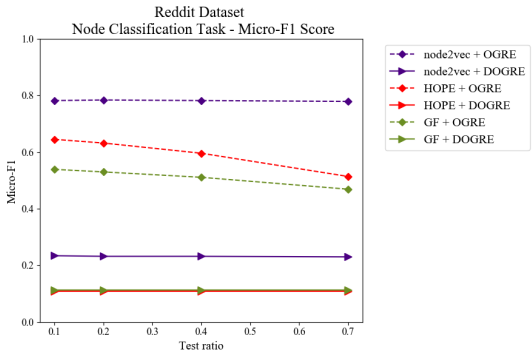
Figure 24: Cora Node Classification



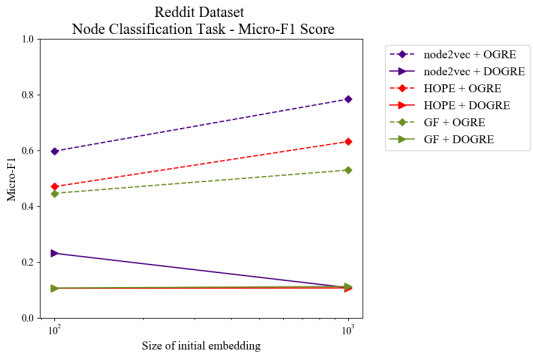
(a) Best Results of Node Classification Macro-F1 score



(b) Initial embedding size affect on Macro-F1 score



(c) Best Results of Node Classification Micro-F1 score



(d) Initial embedding size affect on Micro-F1 score

Figure 25: Reddit Node Classification

8.3 Visualization

8.3.1 OGRE

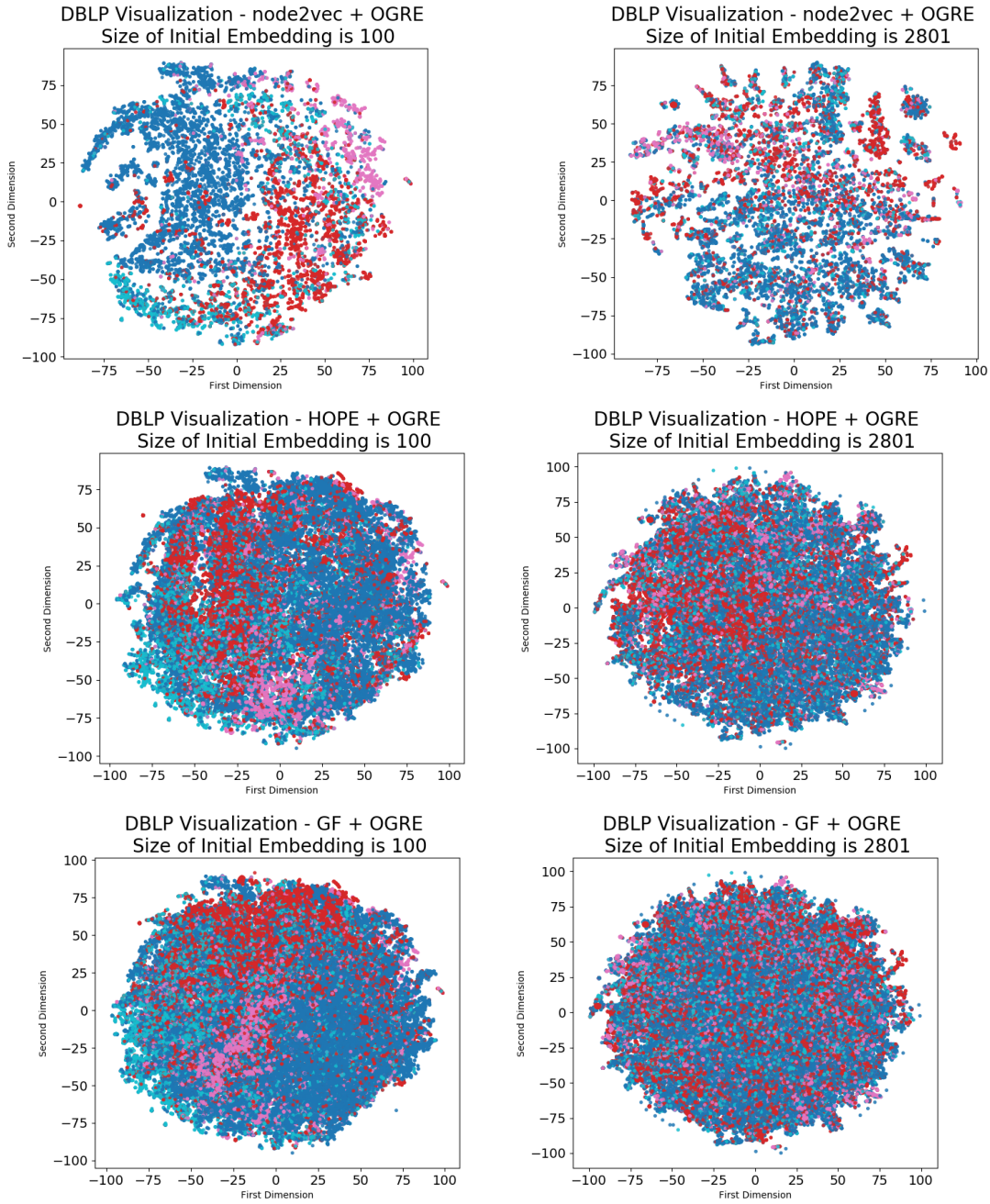


Figure 26: DBLP Visualization OGRE Method

8.3.2 DOGRE

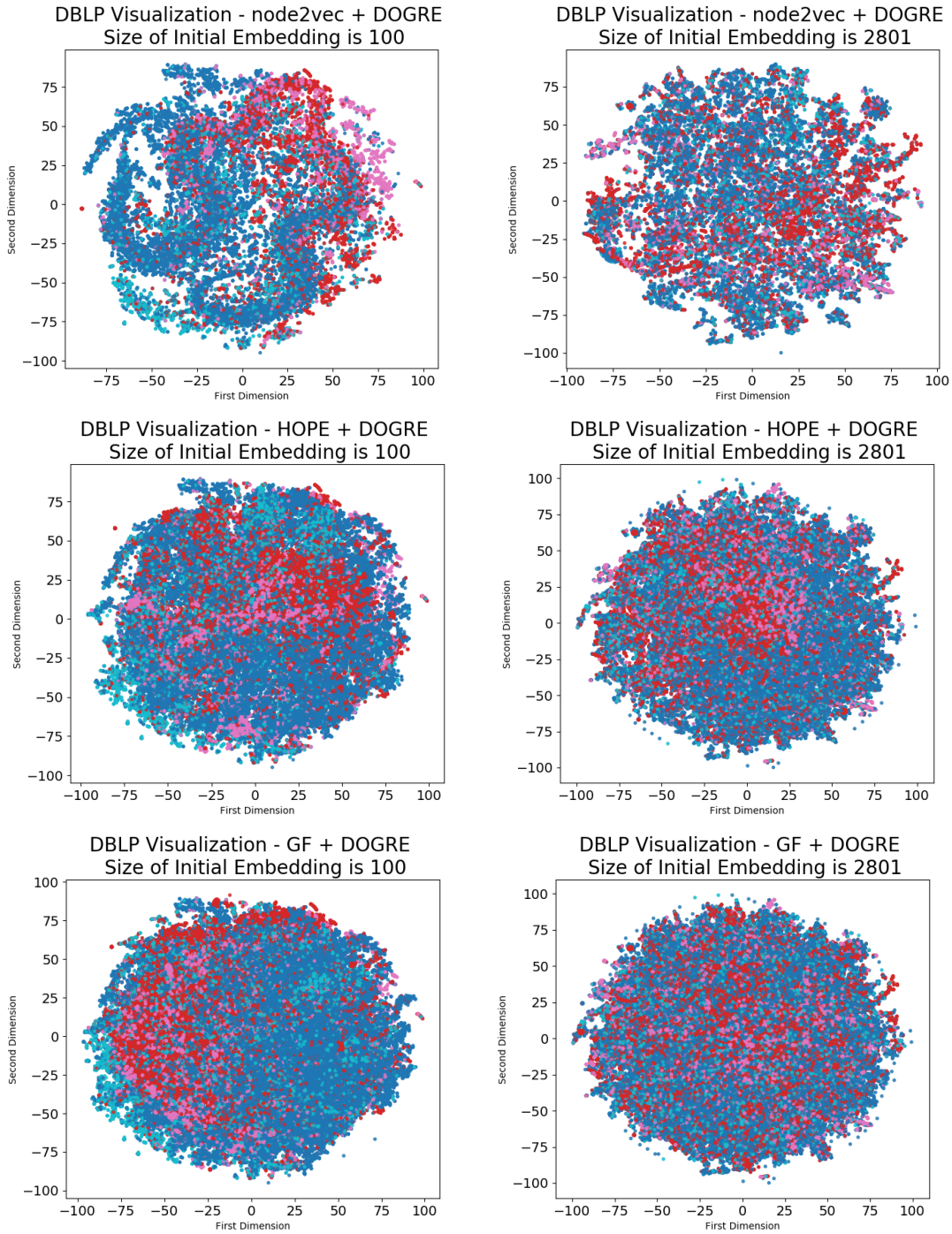


Figure 27: DBLP Visualization DOGRE Method

8.3.3 WOGRE

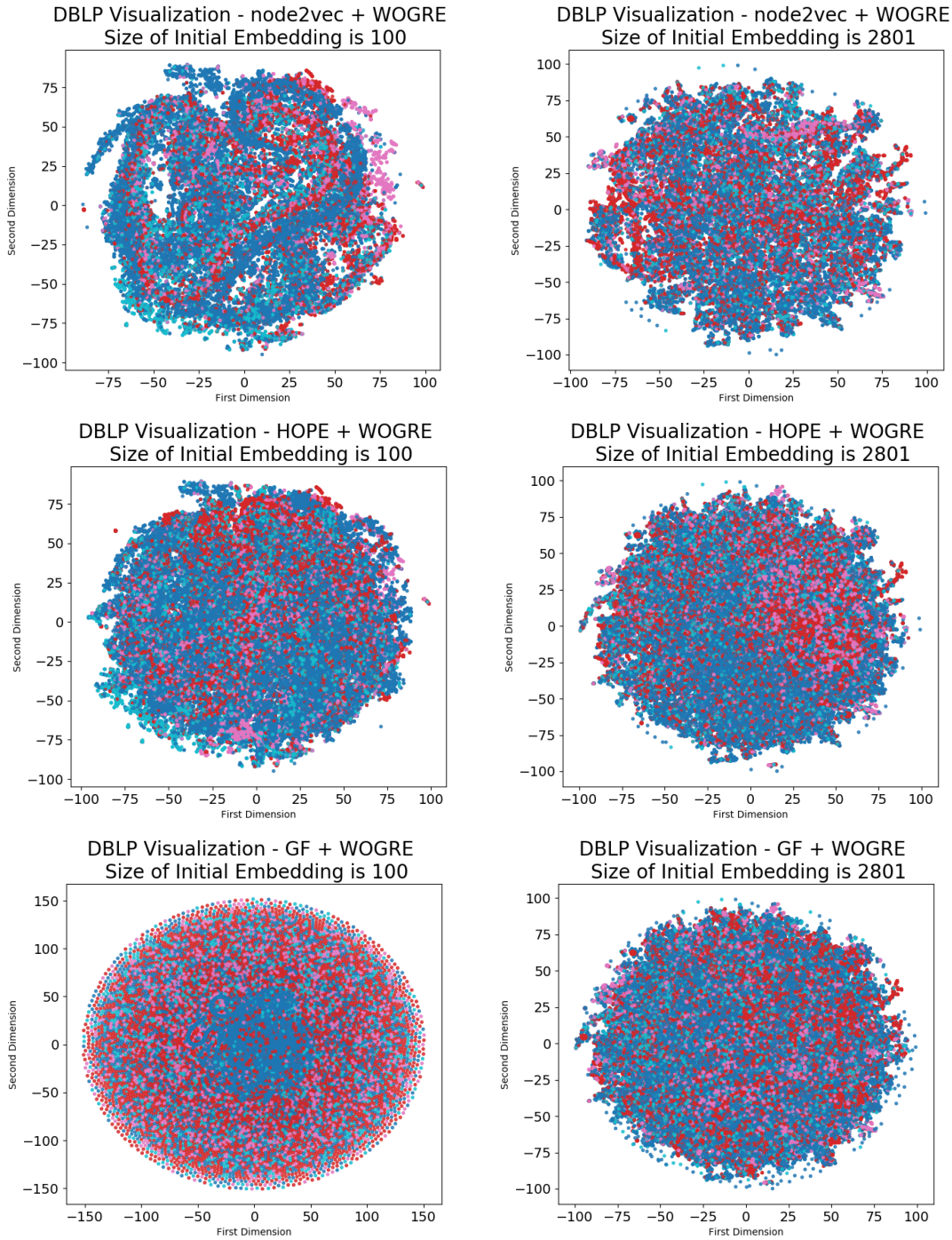


Figure 28: DBLP Visualization WOGRE Method

8.3.4 LGF

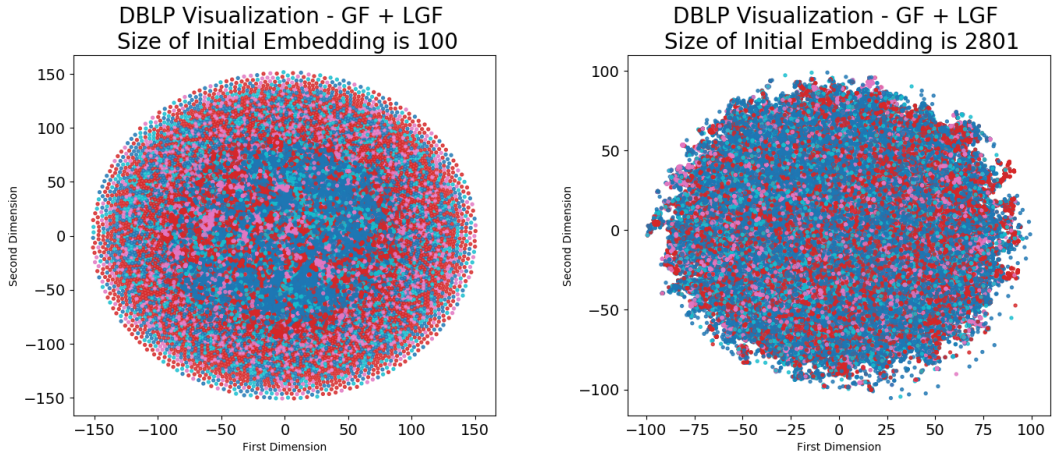


Figure 29: DBLP Visualization LGF Method

References

[1] Shravan Narayanamurthy Vanja Josifovski Alexander J. Smola Amr Ahmed, Nino Shervashidze. Distributed large-scale natural graph factorization. *Proceedings of the 22nd International Conference on World Wide Web, ACM*, pages 37–48, 2013.

[2] Weiqing Wang Bhagya Hettige, Yuan-Fang Li and Wray Buntine. Gaussian embedding of large-scale attributed graphs. *arXiv:1912.00536v1*, 2019.

[3] Steven Skiena Bryan Perozzi, Rami Al-Rfou. Deepwalk: Online learning of social representations. *Proceedings 20th international conference on Knowledge Discovery and data mining*, pages 701–710, 2014.

[4] Wenwu Zhu1 Daixin Wang, Peng Cui. Structural deep network embedding. *Proceedings 22nd international conference on Knowledge Discovery and data mining, ACM*, pages 1225–1234, 2016.

[5] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, pages 78–94, 2018.

[6] Aditya Grover and Jure Leskovec. node2vec: scalable feature learning for networks. *Proceedings of 22nd International Conference on Knowledge Discovery and Data Mining*, ACM:885–864, 2016.

[7] Ajitesh Srivastava Rajgopal Kannan Viktor Prasanna Hanqing Zeng, Hongkuan Zhou. Accurate, efficient and scalable graph embedding. *arXiv:1810.11899v2*, 2018.

[8] Ajitesh Srivastava Rajgopal Kannan Viktor Prasanna Hanqing Zeng, Hongkuan Zhou. Graphsaint: Graph sampling based inductive learning method. *ICLR 2020 Conference Blind Submission*, 2020.

[9] Hao Ma Jian Li Chi Wang Kuansan Wang Jiezhong Qiu, Yuxiao Dong and Jie Tang. Netsmf: Large-scale network embedding as sparse matrix factorization. *International World Wide Web Conference Committee, Creative Commons CC-BY 4.0 License*, 2019.

[10] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ICLR 2017*, 2017.

[11] Tian Ye Guo Xiuyan Lu Haonan, Seth H. Huang. Graph star net for generalized multi-task learning. *arXiv:1906.12330v1*, 2019.

[12] P.Niyogi M.Belkin. Laplacian eigenmaps and spectral techniques for embedding and clustring. *NIPS*, 14:585–591, 2001.

[13] Jian Pei Ziwei Zhang Mingdong Ou, Peng Cui and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM:1105–1114, 2016.

[14] Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, pages 2323–2326, 2000.

[15] Qiongkai Xu Shaosheng Cao, Wei Lu. Grarep: Learning graph representations with global structural information. *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 891–900, 2015.

[16] Qiongkai Xu Shaosheng Cao, Wei Lu. Deep neural networks for learning graph representations. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI Press*, pages 1145–1152, 2016.

[17] Adriano Veloso Nivio Ziviani Tiago Pimentel, Rafael Castro. Efficient estimation of node representations in large graphs using linear contexts. *International Joint Conference on Neural Networks*, 2019.

[18] Jure Leskovec William L. Hamilton, Rex Ying. Inductive representation learning on large graphs. *31st Conference on Neural Information Processing Systems (NIPS)*, 2017.