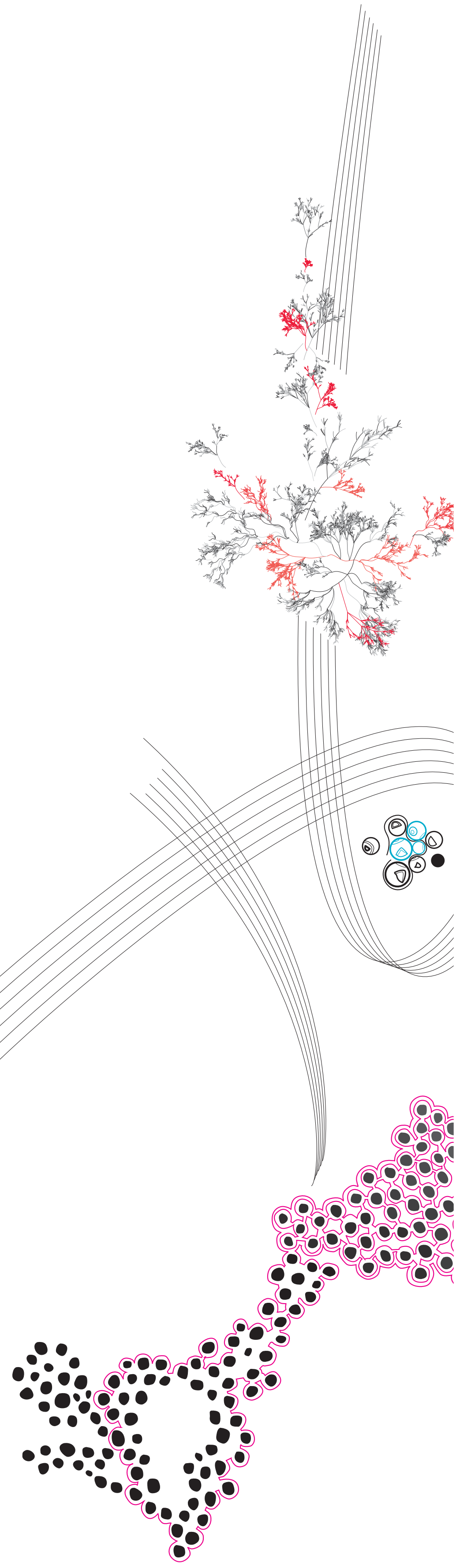
MSc Thesis Applied Mathematics



Optimising node2vec in Dynamic Graphs Through Local Retraining

Michail Angelos Goulis

Supervisor: dr. Clara Stegehuis

August, 2024

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Acknowledgements

I would like to thank my supervisor dr. Clara Stegehuis for her excellent guidance and support during my research. Her expertise and constructive feedback have been essential to shaping my thesis. Moreover, I would like to express my appreciation to the other members of my graduation committee, dr. Mark Uetz and dr. Sophie Langer, for taking the time to read and evaluate my thesis. Next, I would like to thank study advisor Lilian Spijker who was always happy to help me and answer my questions during my thesis and studies in general. Last but not least, I am deeply thankful to my family for their endless love and support and to my friends for their constant motivation throughout my studies.

Contents

Optimising node2vec in Dynamic Graphs Through Local Retraining

Michail Angelos Goulis

August, 2024

Abstract

In network representation learning, deep learning techniques have been on the rise recently due to advancements in neural networks. One of the earliest techniques is node2vec, which generates vector representations of nodes in a graph by simulating random walks to capture network topology and node similarities. However, the time complexity of these techniques is high and scales with the size of the networks. This is particularly important in the case of dynamic networks, which evolve over time and appear in a variety of real-world scenarios. There is still no general method to optimize node2vec for dynamic graphs. In this thesis, we introduce a new technique that makes representation learning computationally feasible in large dynamic graphs. We approximate node2vec on dynamic graphs by focusing on the retraining of local areas affected by graph updates instead of retraining on each graph iteration. We show that this approximating implementation of node2vec results in minimal loss of accuracy while achieving significant speedup, making it a strong optimization technique for use in dynamic graphs.

Keywords: network embeddings, dynamic graphs, word2vec, node2vec, skip-gram, negative sampling, dynamic graphs, extend, prune, centrality

1 Introduction

Many real-world problems involving large networks have increasingly relied on network representation learning techniques. Traditionally, researchers in network science have used specific graph heuristics to extract features such as degree statistics, clustering coefficients, node centrality, etc. However, recent years have seen a surge in approaches that automatically learn to encode network structures using deep learning techniques. The main advantage of these methods is their ability to transform nodes into vectors and matrices, which serve as the primary input for deep learning models. This transformation allows these approaches to fully utilize the knowledge and techniques developed for methods that process matrix-like structures. In addition, these network representation approaches have led to impressive results in various network-based tasks such as node classification, node clustering, and link prediction.

For example, in link prediction, algorithms like Graph Autoencoders and Variational Graph Autoencoders have shown success in predicting future connections in social networks and recommending friends or products [Kipf and Welling, 2017] [Salha et al., 2021]. Also, in node classification, methods like Graph Convolutional Networks (GCNs) have been used to predict the roles or categories of nodes in social networks and citation networks [Kipf and Welling, 2017].

In a similar context, word2vec has been a pioneer in natural language processing (NLP). Its main idea is to employ a neural network to learn dense, continuous vector representations of words from large text corpora. These vectors capture both syntactic and semantic relationships between words, enabling meaningful linguistic analogies through vector arithmetic. A popular example is the analogy "king - man + woman = queen". The encoding of words into vectors has inspired a wave of innovation in various domains, including graph representation learning.

In graph representation learning, nodes are represented as low-dimensional vectors instead of words. One notable graph embedding technique, used primarily for node classification and link prediction, is node2vec, introduced by Grover and Leskovec [Grover and Leskovec, 2016]. This method seeks to broaden the range of graph representation and deep learning techniques, and it will be the main focus of our research.

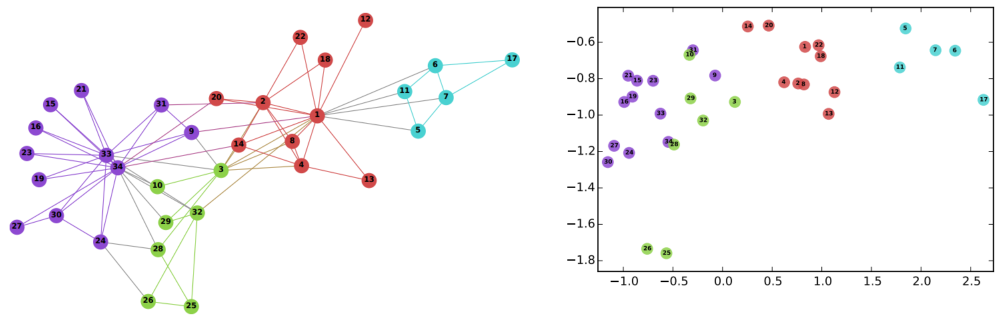


Figure 1: Visualization of a graph and its corresponding embeddings [Perozzi et al., 2014].

1.1 Problem Statement

As stated, node2vec has been a very successful technique in network representation learning, effectively capturing low-dimensional embeddings of nodes for various tasks. However, it can be computationally expensive, even with common optimization techniques (which we analyze further in the next chapters). This issue is especially pronounced in dynamic graphs. A dynamic graph is a graph that is modified by adding new nodes/edges (extending) or removing nodes/edges (pruning). Dynamic graphs appear in various real-time changing phenomena, such as social network interactions, communication networks, and evolving biological systems [Holme and Saramäki, 2012].

For example, consider the case of a social network. When a user adds a new friend, a new edge is created between the nodes representing these two users, extending the graph. When a user unfriends someone, the edge between their corresponding nodes is removed, pruning the graph. In this example, the social network is modelled as a dynamic graph that evolves rapidly over time.

Re-training node2vec on dynamic graphs is inefficient as it requires processing the entire training data again, which is particularly problematic when graph modifications are relatively small compared to the original graph. More specifically,whenever a new node or edge is added to the graph, the existing node embeddings must be updated to reflect these changes. Therefore, in cases where the graph is updated often, this method becomes impractical. An efficient method for calculating the updated embeddings can prove to be highly beneficial in real-world scenarios.

To address this, we propose the following **research questions**:

RQ1 How can we make the node2vec algorithm more efficient in dynamic graphs?

RQ2 How does approximating node2vec in dynamic graphs affect accuracy and training time?

RQ3 How is accuracy affected depending on the type of dynamic update (removing nodes randomly or according to a specific graph statistic)?

We investigate the efficiency of a custom node2vec implementation in node classification for local retraining compared to global retraining. Specifically, our approach focuses on retraining only the affected areas of the graph after updates, rather than retraining the entire network. This localized retraining aims to reduce computational complexity and increase efficiency while maintaining accuracy.

1.2 Thesis Outline

The thesis is structured as follows. In Chapter 2 we introduce the theory behind *word2vec* to get to the basis of *node2vec*. Chapter 3 defines the evaluation metrics used in our graph models. Following this, Chapter 4 explains our process, the datasets we experimented on as well as the custom implementations. Furthermore, in Chapter 5 we analyse the results of our research. Finally in Chapter 6 we include the conclusion and discussion.

2 Background

In this section we will describe the background of *word2vec*, *node2vec*, graph learning, centrality measures and dynamic networks.

2.1 word2vec

In the field of natural language processing (NLP), *word2vec* is a technique for converting real language words into high dimensional vector representations. It was published by Tomas Mikolov and his colleagues at Google in the paper "Efficient Estimation of Word Representations in Vector Space" [Mikolov et al., 2013].

The core idea behind *word2vec* is its ability to capture the context and meaning of words by embedding them in a continuous vector space. It analyses large amounts of text data, and learns to group word sets that appear frequently together. As a result, it can create meaningful semantic relationships which can be used for various tasks such as sentimental analysis, machine translation etc.

Two primary architectures underpin the *word2vec* framework: the continuous bag-of-words (CBOW) model and the skip-gram (SG) model. These architectures represent distinct approaches to learning word embeddings, each with its own strengths and trade-offs. We discuss the latter as it is also used in the training of *node2vec*.

2.1.1 Skip-gram

Skip-gram is a learning model for *word2vec* responsible for learning the word embeddings. Training of the skip-gram model involves utilizing efficient lookup tables and dot products rather than dense matrix multiplications as found in other neural network architectures [Mikolov et al., 2013]. Its training objective is to maximise the probability of observing the actual words given the target word, which is the word being predicted.

Given a word sequence, $w_1, w_2, \ldots, w_n$, for training, the skip-gram model seeks to maximise the following objective in order to learn the embeddings

$$L_{SG} = \frac{1}{n} \sum_{i=1}^{n} \sum_{\substack{|j|<c \\ j\neq 0}} \log p(w_{i+j}|w_i), \tag{1}$$

where w_i is a target word and w_{i+j} is a context word within a window of size c [Goldberg and Levy, 2014]. Specifically, the target word is the word being predicted, while the context words are the words that surround it.

We define the softmax unit function

$$p(w_{i+j}|w_i) = \frac{\exp(\mathbf{t}_{w_i} \cdot \mathbf{c}_{w_{i+j}})}{\sum_{w\in W} \exp(\mathbf{t}_{w_i} \cdot \mathbf{c}_w)},$$

where $p(w_{i+j}|w_i)$ represents the probability that w_{i+j} appears given w_i, $\mathbf{t}_w$ and $\mathbf{c}_w$ are w's embeddings when it behaves as a target and context, respectively. W represents the vocabulary set [Kaji and Kobayashi, 2017]. It could be seen as the generalisation of the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$. More specifically, the sigmoid function can be seen as a special case of the softmax function when dealing with two classes of a binary classification problem. The softmax function works well as when the word vectors are near each other, their dot product increases, leading to a higher numerator and thus higher probabilities. This allows the model to give more weight on similar words which appear close to each other.

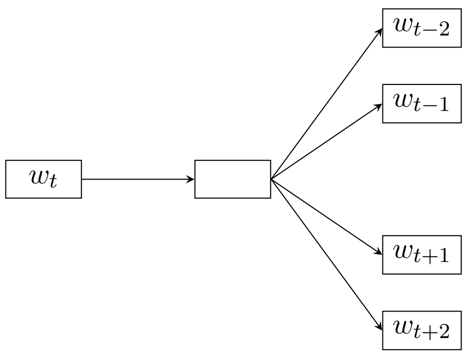


FIGURE 2: Skip-gram model architecture. Adapted from [Mikolov et al., 2013].

An alternative more abstract formulation of the problem is found in [Goldberg and Levy, 2014]. In the skip-gram model, given a corpus of words w and their contexts c, we consider the conditional probabilities $p(c|w)$ of observing a context word c given w and a corpus text. We define parameters θ as v_c, v_w, for $w \in V, c \in C, i \in 1, \ldots, d$ (a total of $|C| \times |V| \times d$ parameters) where v_c and $v_w \in \mathbb{R}^d$ are vector representations for c and w respectively, and a set C of all available contexts. Our goal is to set θ to maximise

$$\arg\max_{\theta} \prod_{(w,c)\in D} p(c|w;\theta), \tag{2}$$

where D is the set of all word and context pairs we extract from the text.

The conditional probability $p(c|w;\theta)$ is defined as

$$p(c|w;\theta) = \frac{\exp(v_c \cdot v_w)}{\sum_{c' \in C} \exp(v'_c \cdot v_w)},$$

This alternative definition uses simpler and more clean notation of the target and context words and emphasizes the importance of the embeddings via the optimal parameter θ. However, one can notice from both definitions that computing the sum in the denominator over all the contexts c' is expensive therefore we apply *negative sampling.*

2.1.2 Negative Sampling

Negative Sampling is an efficient technique used in training of *word2vec* to derive word embeddings [Mikolov et al., 2013]. Instead of including all possible words (w, c) as shown in (2), only some negative examples (not correct pairs of target-context words) are sampled. Negative-sampling is based on the skip-gram model but it optimises a different objective. We derive the negative-sampling objective, which aims to maximize the likelihood of observed word-context pairs while minimizing the likelihood of randomly sampled negative examples, as shown in [Goldberg and Levy, 2014].

We consider a pair (w, c) of word and context and denote by $p(D = 1|w, c; \theta)$ the probability that this pair comes from the training data with parameters θ controlling the distribution.

Our goal is to find parameters that maximise the probabilities that all the observations indeed came from the data and by taking the logarithm of this objective we have

$$\arg\max_{\theta} \sum_{(w,c)\in D} \log p(D = 1|w, c; \theta) \tag{3}$$

We then define $p(D = 1|w, c; \theta) = \frac{1}{1+\exp(-v_c \cdot v_w)}$ leading to the objective

$$\arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-v_c \cdot v_w)}$$

This objective has a trivial solution by setting θ such that $v_c = v_w$ and $v_c \cdot v_w = K$ for all v_c, v_w, where K is a large enough number.

To prevent all the vectors from having the same value, we can disallow some (w, c) combinations. A way to achieve that is by including some (w, c) pairs for which $p(D = 1|w, c; \theta)$ is low, that is pairs which are not in the data. We generate the set D' of random incorrect (w, c) pairs, that is pairs of words that are not observed together in the training data, to serve as negative samples for training the model (the name "negative-sampling" stems from the set D' of randomly sampled negative samples). The optimisation objective (3) becomes now

$$\begin{aligned}
&\arg\max_{\theta} \prod_{(w,c)\in D} p(D = 1|c, w; \theta) \prod_{(w,c)\in D'} p(D = 0|c, w; \theta) \\
&= \arg\max_{\theta} \prod_{(w,c)\in D} p(D = 1|c, w; \theta) \prod_{(w,c)\in D'} \big(1 - p(D = 1|c, w; \theta)\big) \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log p(D = 1|c, w; \theta) + \sum_{(w,c)\in D'} \log \big(1 - p(D = 1|c, w; \theta)\big) \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-v_c \cdot v_w)} + \sum_{(w,c)\in D'} \log \frac{1}{1 + \exp(v_c \cdot v_w)} \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log \sigma(v_c \cdot v_w) + \sum_{(w,c)\in D'} \log \sigma(-v_c \cdot v_w)
\end{aligned}$$

A more abstract version of the objective can be found in [Mikolov et al., 2013] defined as

$$\log \sigma(v'^{T}_{w_O} v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)}[\log \sigma(-v'^{T}_{w_i} v_{w_I})], \tag{4}$$

where v_{w_I} is the vector representation of the input word w_I, v'_{w_O} is the representation of the output word w_O, v'_{w_i} is the representation of negative sample word w_i, $P_n(w)$ denotes the noise distribution from which negative samples w_i are drawn and $\mathbb{E}_{w_i \sim P_n(w)}$ denotes the expectation with respect to the noise distribution.

Initially, the unigram distribution was employed as noise distribution, where probabilities are assigned to words based on their occurrence frequency in the corpus. Words with higher frequency have also higher probability of being sampled. Subsequently, a tailored approach was adopted, where each word is sampled proportional to its unigram distribution raised to the power of $\frac{3}{4}$. This adjustment was found to outperform both the standard unigram and uniform distribution in accuracy of analogical reasoning tasks, that is the ability of understanding the analogous relationships between words (e.g. if the model knows that "man" is to "woman" as "king" is to "queen") [Mikolov et al., 2013].

2.2 node2vec

Building on the principles of *word2vec*, Grover and Leskovec introduced *node2vec*, a graph-based learning algorithm used to generate embeddings for nodes in a graph [Grover and Leskovec, 2016]. It groups together node embeddings with similar characteristics (depending on the analysis) while preserving the structural properties of the network. Unlike traditional approaches that solely rely on labelled data, *node2vec* operates in a semi-supervised manner, effectively utilizing both labelled and unlabelled data to enhance its learning process.

The primary objective of *node2vec* is to represent each node in the graph as a dense vector, capturing its semantic and topological properties. These embeddings serve as feature representations that can be fed into downstream machine learning models or neural networks for various tasks such as node classification, link prediction, and graph visualization.

2.2.1 Skip-gram

Similarly to word2vec, skip-gram is used as the learning model while training. The equivalent optimisation to the objective (3) for node2vec is

$$\max_f \sum_{u \in V} \sum_{n_i \in N(u)} \log P(n_i|u; f) \tag{5}$$

In the network context, for each node u a d dimensional vector is learned and represented as $f(u)$ where f is the embedding function, V is the set of all nodes and $N(u)$ represents the *walk neighbourhood* of node u which is obtained through a series of specifically defined random walks originating from u (see Chapter 2.2.3). The way the neighbourhood $N(u)$ of a node is defined is not by the standard method of including the adjacent nodes, but rather by incorporating nodes that are generated from a random walk.

The probability $P(n_i|u; f)$ is modelled by a softmax unit [Armandpour et al., 2019]:

$$P(n_i|u; f) = \frac{\exp\left(f(n_i)^T \cdot f(u)\right)}{\sum_{v \in V} \exp\left(f(v)^T \cdot f(u)\right)}$$

2.2.2 Negative Sampling

Similarly to word2vec, negative sampling in node2vec is used to overcome the computational infeasibility of the skip-gram objective because each of the softmax terms requires summation over all vertices [Yang et al., 2020]. Negative sampling provides a computationally feasible approximation to the objective (5) by replacing each $\log P(n_i|u; f)$ term with

$$\log\left(\sigma(f(n_i)^T \cdot f(u))\right) + \sum_{j=1}^{k} \mathbb{E}_{v_j \sim P(v)}[\log \sigma(-f(v_j)^T \cdot f(u))]$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ and $P(v)$ is a distribution proportional to d_v^α, where d_v is the degree of node v [Mikolov et al., 2013], $\alpha = 0.75$ and k is the number of negative samples.

Regarding the sampling of nodes, positive examples are pairs of nodes that are connected in the graph and negative examples are pairs that are not connected. Instead of considering all possible negative examples, negative sampling randomly samples a small number of negative examples during each training iteration.

Since the sampling is proportional to the graph's degree distribution $P(v)$, nodes with more connections are more likely to be chosen as negative samples. This is because high-degree nodes have a higher chance of appearing in random walks by chance, even if they aren't truly connected to the central node in the context of the specific task.

2.2.3 Biased Random Walks

At its core, the *node2vec* algorithm explores the notion of node similarity within the graph by employing a biased random walk strategy. In that sense, it is similar to *word2vec* as it treats nodes as words and random walks as sentences. However, it expands on the one-dimensional nature of *word2vec*, as sentences are linear sequences, whereas the walk neighbourhood is a more complex, two-dimensional structure. By intelligently navigating the graph space, it ensures that nodes with similar network neighbourhoods are embedded closely together in the vector space. This strategy enables the embeddings to preserve the local and global structural characteristics of the original graph.

One of the key challenges in designing *node2vec* lies in balancing the exploration and exploitation trade-off during the random walk process. The algorithm incorporates two parameters, namely the return and in-out parameters p and q, to control the exploration behaviour, allowing it to capture both **homophilic** and **structural equivalence** among nodes [Grover and Leskovec, 2016].

More specifically:

- **Homophily**: Nodes are organised based on communities they belong to (e.g. center node connected to 4 surrounding it, like 5 in a dice)
- **Structural equivalence**: Nodes share structural roles e.g. two hubs (nodes that are two distinct communities but have a hub role)

The way of sampling nodes can create different structures when trying to find the nodes of a neighbourhood in a random walk. In general, there are two main sampling paradigms:

- **Breadth-first Sampling (BFS)**: This algorithm explores a graph systematically, visiting all the neighbouring nodes of the current node before moving to the next level. It focuses on exploring the neighbourhood of a node and therefore discovers other nodes that share similar connections (structural equivalence).
- **Depth-first Sampling (DFS)**: This algorithm explores a graph by following a single path as deep as possible until it reaches a dead end (no unvisited neighbors). Then, it backtracks and explores another branch. It samples nodes sequentially at increasing distances from the source node, making it more likely to encounter node that share similar characteristics. It tends to stay within clusters of similar nodes (homophily).

We simulate a random walk of length l for a given source u. Let c_i denote the ith node in the walk with $c_0 = u$. The nodes c_i are generated by

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where π_{vx} is the unnormalised transition probability between nodes v and x and Z is the normalising constant [Grover and Leskovec, 2016].

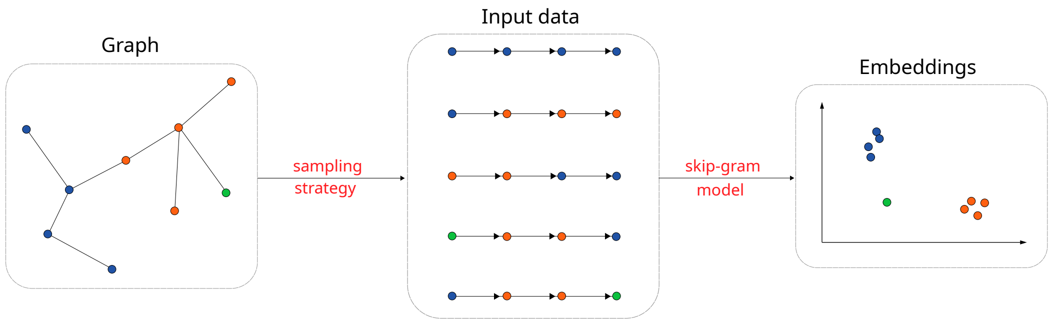


FIGURE 3: node2vec embedding process. Adapted from [Berbatova, 2020].

To introduce bias in random walks, we define second-order walks and incorporate two parameters: p and q. Second-order walks consider both the current and previous states. Essentially, when the algorithm determines traversal probabilities, it also takes into account the previous step.

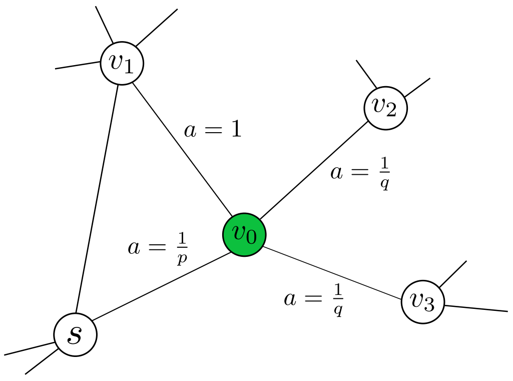


FIGURE 4: Transition probability diagram. Adapted from [Grover and Leskovec, 2016].

Consider a random walk that just traversed edge (s, v_0) and now resides at node v_0 (Figure 4). The walk now needs to decide on the next step so it evaluates the transition probabilities π_{v_0x} on edges (v_0, x) leading from v_0.

We set the transition probability π_{v_0x} to

$$\pi_{v_0x} = \begin{cases} \frac{1}{p} & \text{if } d_{sx} = 0 \\ 1 & \text{if } d_{sx} = 1 \\ \frac{1}{q} & \text{if } d_{sx} = 2 \end{cases} \tag{7}$$

The **return parameter** p controls the likelihood of backtracking and revisiting a node. A high value of p reduces the chances of revisiting nodes, preventing 2-hop redundancy and promoting moderate graph exploration. Conversely, a low p value increases the likelihood of backtracking, keeping the walk closer to the starting node.

The **in-out parameter** q allows differentiation between inward and outward nodes during traversal. A high q value ($q > 1$) biases the walk towards nodes near the previous step, providing a local view of the graph similar to a **BFS**. In contrast, a low q value

($q < 1$) encourages the walk to visit nodes further away, promoting outward exploration akin to a **DFS**.

In the example of Figure 5, we observe that from node v_0 we explore s_1, s_2, s_3, s_4 via BFS as the random walks stay around the central starting node, generating the neighborhood $N_{BFS}(v_0) = \{s_1, s_2, s_3, s_4\}$. On the other hand, DFS traverses nodes s_4, s_5, s_6 in a path, generating $N_{DFS}(v_0) = \{s_4, s_5, s_6\}$.

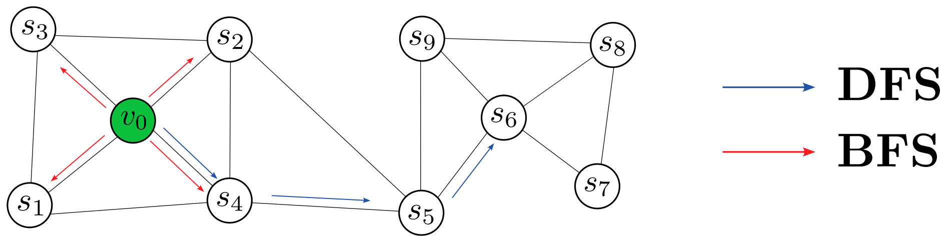


FIGURE 5: DFS and BFS diagram. Adapted from [Grover and Leskovec, 2016].

2.3 Node Removal Strategies

Referring back to RQ3 of the introduction, it is interesting to explore how the strategy of removing nodes affects the accuracy of a node2vec model. We focus on removing nodes randomly and based on **centrality** measures. In network analysis, centrality is defined as an indicator of ranking nodes based on their position and connections relative to other nodes.

2.3.1 Betweenness Centrality

Betweenness centrality is a measure of centrality based on shortest paths. It represents the degree to which nodes stand between each other. Betweenness centrality was introduced by Linton C. Freeman in his paper "A Set of Measures of Centrality Based on Betweenness" published in 1977 [Freeman, 1977].

Betweenness centrality of node v is the total of the fraction of shortest paths that pass through v. In other words is calculated by counting how often a node appears on the shortest paths between other nodes in the networks.

$$C_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

where V is the set of nodes, $\sigma(s,t)$ is the number of shortest (s,t)-paths, and $\sigma(s,t|v)$ is the number of shortest (s,t)-paths that pass through v where $v \notin \{s,t\}$. If $s = t, \sigma(s,t) = 1$ and if $v \in \{s,t\}, \sigma(s,t|v) = 0$ [Brandes, 2008].

2.3.2 Degree Centrality

An even simpler metric of centrality is **degree centrality**. It is defined as the number of connections a node has to other nodes. More specifically, the degree centrality of node v is defined as

$$C_D(v) = deg(v)$$

2.4 Related Work

In this section, we briefly discuss related work on embedding models for both static and dynamic graphs. The DeepWalk algorithm is the first work that uses a standard random walk to create node sequences, namely the lists of traversed nodes, from a static network [Perozzi et al., 2014]. Afterwards, node2vec was introduced as a modified version of DeepWalk, utilizing BFS and DFS on random walks [Grover and Leskovec, 2016].

There have also been many papers explaining incremental skip-gram with negative sampling in word2vec but not for node2vec. One of the most notable contributions is by [Goldberg and Levy, 2014], also referenced previously in our analysis, where they aim to explain the main negative sampling equation found in [Mikolov et al., 2013]. Other papers focus on hyperparameter tuning. For example, the paper from [Caselles-Dupré et al., 2018] focuses on finding optimal hyperparameters in a recommendation system setting, employing grid search on various datasets to enhance performance.

There have not been many studies focusing on the performance of node2vec in large scale dynamic networks or ways of optimising its performance. The original paper from Grover and Leskovec analyses node2vec performance on popular graphs and includes some experimental results in static graphs [Grover and Leskovec, 2016]. In addition, the paper from Peng et al. [Peng et al., 2020] proposes an efficient incremental skip-gram algorithm with negative sampling for dynamic network embeddings and includes theoretical analyses about the performance guarantee. However, their proposed objective function appears to contain a conceptual error, which could potentially impact the validity of their performance guarantees and the overall correctness of the algorithm.

There are also more recent studies that focus on embedding dynamic graphs and are not based on *node2vec*. One of them is from Trivedi et al. [Trivedi et al., 2019] where they model the occurrence of an edge as a point process and parametrise the intensity function by using a neural network, taking node embeddings as the input. They claim that their approach outperforms representative baselines, including *node2vec*, for the problem of dynamic link prediction and event time prediction. However, their method evaluates only on two datasets which does not seem representative enough.

Another type of research that does not rely on node embeddings at all is from Pareja et al. [Pareja et al., 2020]. They propose the *EvolveGCN*, a model that adapts graph convolutional networks (GCNs) over time using a recurrent neural network (RNN) to evolve GCN parameters, The model is evaluated on link prediction, edge classification, and node classification.

3 Evaluation

In this chapter, we introduce the evaluation metrics used to assess the performance of our custom *node2vec* implementation in node classification.

3.1 One-vs-Rest(OvR)

One-vs-Rest(OvR), also known as one-vs-all, is a strategy used for multiclass classification tasks where you have more than two classes. In simple terms, it is the generalisation of logistic regression to multiple classes. In this approach, multiple binary classifiers are trained, each one distinguishing between one class and the rest of the classes. More specifically, separate binary classification models where each model is trained with positive samples belonging to the class of interest and samples from all other classes act as negative examples. To predict the class label for a new sample, the sample is passed through every binary logistic regression model and the model that gives the highest probability or confidence score is chosen. The final output of the OvR logistic regression classifier is the predicted class label for each sample [Bishop, 2006].

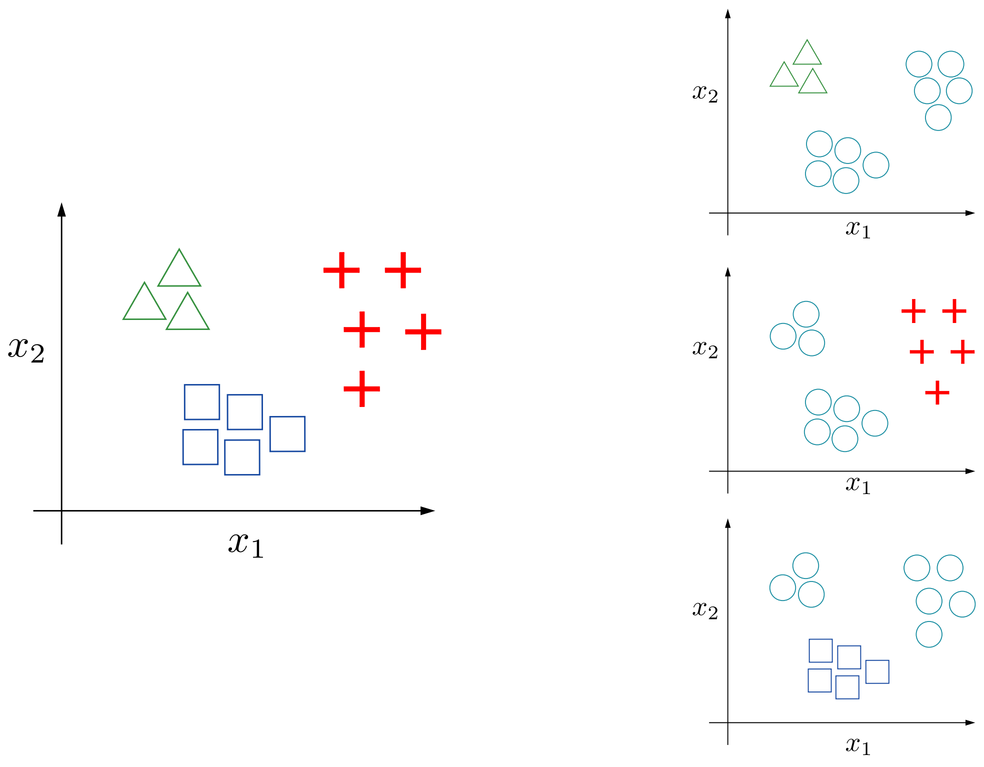


FIGURE 6: One-vs-Rest.

3.2 F-score

To define F-score, we first need to define accuracy, precision, and recall. Accuracy is a measure of the overall correctness of the classifier, calculated as the ratio of correctly predicted instances to the total instances in the dataset

$$Accuracy = \frac{TP + TN}{P + N} = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision (or positive prediction value), on the other hand, focuses on the relevance of the model's predictions, measuring the ratio of correctly predicted positive observations to the total predicted positives

$$Precision = \frac{TP}{TP + FP}$$

Recall (or true positive rate) gauges the classifier's ability to identify all relevant instances, calculated as the ratio of correctly predicted positive observations to all actual positives in the dataset

$$Recall = \frac{TP}{TP + FN}$$

The F-score, or F1-score, harmonizes precision and recall into a single metric, providing a balance between these two crucial measures. It is defined as the harmonic mean of precision and recall, given by the formula

$$F_1 = \frac{2}{Recall^{-1} + Precision^{-1}} = 2\frac{Precision \cdot Recall}{Precision + Recall} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

This metric is particularly useful in situations where there is an uneven class distribution, as it considers both false positives and false negatives. The F-score ranges from 0 to 1, where 1 indicates perfect precision and recall, and 0 represents the worst possible F-score, typically seen when precision or recall is 0 [Taha and Hanbury, 2015].

Computing *precision* and *recall* to calculate the *F-score* proves useful in a binary class classification task. However, in a multi-class setting we would need to calculate the *F-score* in a One-Vs-Rest (OvR) approach. In this OvR approach, we determine the metrics for each class separately, as if there is a different classifier for each class. With this approach, we obtain multiple per-class *F-scores*. It would be beneficial to average them to end up with a single number to describe overall performance. For that reason, we introduce some averaging methods.

3.3 Micro and Macro Averaging

In a mutilabel classification task, we define *Macro-F1* score as the average of F1-scores throughout all classes

$$Macro - F_1 = \frac{1}{N}\sum_{i=1}^{N} F_{1i} = \frac{1}{N}\sum_{i=1}^{N} 2\frac{Precision_i \cdot Recall_i}{Precision_i + Recall_i}$$

where

$$Precision_i = \frac{TP_i}{TP_i + FP_i}$$

and

$$Recall_i = \frac{TP_i}{TP_i + FN_i}$$

In other words, it computes the F1 score for each class independently and then takes the average. This treats all classes equally regardless of their frequency, making it optimal when a dataset is imbalanced but the classes are equally important

We define *Micro-F1* score as the global average F1 score by aggregating all true positives, false positives, and false negatives across all classes.

$$Micro - F_1 = 2\frac{Micro\ Precision \cdot Micro\ Recall}{Micro\ Precision + Micro\ Recall}$$

where

$$Micro\ Precision = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N}(TP_i + FP_i)}$$

and

$$Micro\ Recall = \frac{\sum_{i=1}^{N} TP_i}{\sum_{i=1}^{N}(TP_i + FN_i)}$$

This makes it particularly useful when measuring the overall performance across all classes [Manning et al.].

4 Method

4.1 Specifications of Datasets

We focus our research on the datasets that are more commonly used in node classification tasks. More specifically:

Name	$\lvert V \rvert$	$\lvert E \rvert$	*Labels*
BlogCatalog	10,312	333,983	39
PPI	3,890	38,739	50
Wikipedia	4,777	92,517	40
Cora	2,708	5,278	7

Table 1: Graph datasets.

- BlogCatalog [Zafarani and Liu, 2009]: This dataset includes 10,312 bloggers represented as nodes and 333,983 friendship relationships represented as edges. Each blogger is associated with labels that represent their topic interests. There are 39 distinct labels in the network, and a single blogger can have multiple labels.

- PPI [Breitkreutz et al., 2007]: This dataset includes 3,890 nodes representing proteins and 38,739 interactions represented as edges. Each protein is associated with labels that represent biological states derived from hallmark gene sets. There are 50 distinct labels in the network.

- Wikipedia [Mahoney, 2011]: This dataset includes 4,777 nodes representing words and 92,517 edges representing co-occurrences between words. Each word is associated with labels that represent its part-of-speech role. There are 40 distinct labels in the network.

- Cora [McCallum et al., 2000]: This dataset includes 2,708 machine-learning papers represented as nodes and 5,429 citations represented as directed edges. Each paper is associated with labels that represent its class. There are 7 distinct classes in the network.

4.2 Algorithmic Implementation

We use eliorc's *node2vec* implementation [eliorc, 2022] in our experiments. Negative sampling in node2vec is implemented using *gensim.word2vec* [Gensim, 2024]. In our research, we adapt the *node2vec* implementation for use in dynamic graphs. Specifically, we take snapshots of the evolving graphs and apply *node2vec* to each snapshot. This approach allows us to simulate a dynamic graph evolving over time.

Additionally, we modified the implementation to support the *starting_ nodes* argument. Specifically, we define an additional argument to initiate random walks from specific starting nodes. The modified *node2vec* function is defined as:

```
node2vec(dimensions, num_walks, walk_length, p, q, starting_nodes)
```

The code implementation is publicly available at `https://github.com/mihalisag/dynamic_graph_learning`.

4.2.1 Dynamic Graph Generation

To dynamically update the graph, we first follow a custom procedure adapted from the paper [Peng et al., 2020] to generate graphs dynamically. More specifically, we:

1. Define the starting connected graph G_n
2. Remove a node while ensuring the updated graph G^* is still connected $G^* = G_{t-1} \setminus v$
3. Store the updated graph
4. Repeat the process for a specified number of nodes r
5. Reverse the graph list

In step 2 of removing a node we have three options. The first option is to choose a node randomly. The second option is to use the betweenness centrality (see Chapter 2.3.1). Calculating betweenness centrality is computationally expensive because it involves finding all shortest paths. To speed up the calculation we use the *nx-parallel* package from [Juneja, 2024] which can work in parallel to calculate the betweenness centrality for multiple nodes simultaneously. In addition, we choose a subset of the nodes to increase efficiency by not having to iterate through the random walks of all possible pairs, while still providing a good approximation. The third option is to use degree centrality (see Chapter 2.3.2) which is also computationally expensive but not as demanding as betweenness centrality.

The reason we have three different node removal methods is to ensure the node removal process is unbiased. Specifically, by removing nodes based on centrality metrics instead of purely at random, we can more effectively analyze the impact of removing key nodes in the graph. Random node removal offers a baseline for comparison. Conversely, removing nodes with high betweenness centrality targets those that act as bridges within the network, highlighting effects on connectivity. Similarly, using degree centrality to eliminate nodes with the most connections emphasizes important changes in the network's structure. Examining these varied strategies helps us understand how different removal methods influence the network.

After this process, we obtain a list of connected graphs starting with a connected subgraph G_0 where $|V_{G_{n-r}}| = |V_{G_n}| - r$, for $r = 0, 1, \ldots n$ and building up to the initial graph G_n.

The main reason we follow this process is to maintain the original structure of the initial graph while we take its subgraphs. This is particularly important for our experiments later in node classification as it is helpful to keep the structure of the dataset used.

We can now use this list of graphs for our extending and pruning experiments. We define the algorithms used in pseudocode.

Algorithm 1: remove_nodes_connected(G, N, removal_process)

```
Data: Initial graph $G$, Number of nodes to remove $N$, Node removal process
      $removal\_process$
Result: Pruned graph $G'$, Removed nodes-edges dictionary $D$
Set random seed to 42;
$G' \leftarrow$ copy of $G$;
$D \leftarrow$ empty dictionary;
$ignore\_list \leftarrow$ empty list;
while $N > 0$ and $|G'.nodes| > 0$ do
    if $removal\_process$ = 'random' then
        $v \leftarrow$ random node from $G'.nodes$;
    else if $removal\_process$ = 'betweenness_centrality' then
        $bet\_centr\_dict \leftarrow$ betweenness centrality of $G'$;
        $bet\_centr\_dict \leftarrow$ filter out nodes in $ignore\_list$;
        $v \leftarrow$ node with max betweenness centrality in $bet\_centr\_dict$;
    else if $removal\_process$ = 'degree_centrality' then
        $deg\_centr\_dict \leftarrow$ degree centrality of $G'$;
        $deg\_centr\_dict \leftarrow$ filter out nodes in $ignore\_list$;
        $v \leftarrow$ node with max degree centrality in $deg\_centr\_dict$;
    $components \leftarrow$ list of connected components of $G'$;
    $component\_v \leftarrow$ None;
    foreach $component \in components$ do
        if $v \in component$ then
            $component\_v \leftarrow component$;
            break;
    if $component\_v \neq None$ then
        $subgraph \leftarrow$ create subgraph of $G'$containing $component\_v$;
        $subgraph \leftarrow$ remove node $v$ from $subgraph$;
        if $subgraph$ is connected then
            $removed\_edges \leftarrow$ edges of $G'$containing $v$;
            add $(v, removed\_edges)$ pair to $D$;
            $G' \leftarrow$ remove node $v$ from $G'$;
            $N \leftarrow N - 1$;
        else
            if $removal\_process \in$ {'betweenness_centrality', 'degree_centrality'}
             then
                add $v$ to $ignore\_list$;
return $G', D$
```

Algorithm 2: dynamic_graph_gen(G, R)

```
Data: Initial graph $G$, Number of different nodes $R$
Result: List $dynamic\_graphs$ of dynamic graphs
$dynamic\_graphs \leftarrow [G]$;
$G' \leftarrow$ copy of $G$;
while $i \leq R$ do
    $G' \leftarrow$ remove_nodes_connected$(G', 1)$;
    append $G'$ to $dynamic\_graphs$;
    $i \leftarrow i + 1$;
return $dynamic\_graphs$
```

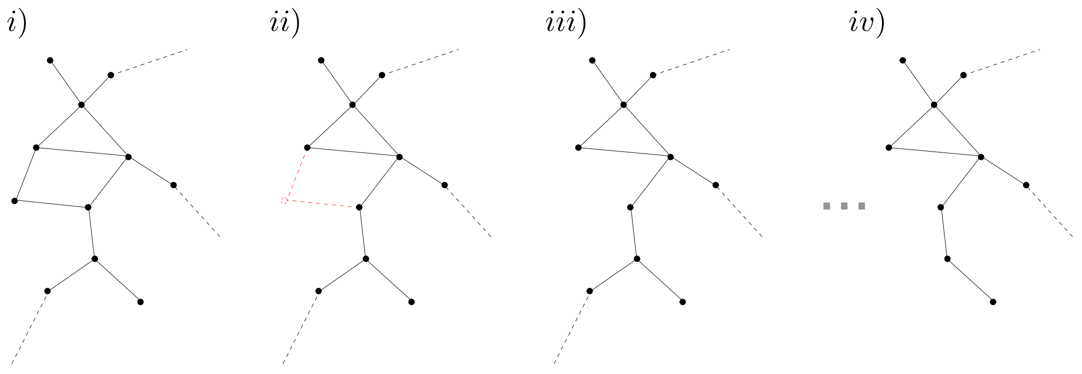


FIGURE 7: Dynamic graph generation. (i) Initial graph structure. (ii) Selection of a node for removal while ensuring the graph remains connected. (iii) Removal of the selected node. (iv) Final resulting graph.

We also define some of the variables that are used in both extending and pruning algorithms:

- X: list of vector embeddings
- y: group label of the corresponding embeddings of X
- D: dictionary of (node label, vector embedding) pairs
- $model \leftarrow node2vec(G, \text{starting\_nodes})$: model by applying `node2vec` to graph G and initiating random walks from starting_nodes (optional)
- emb_group($model.fit()$): fit model and generate X, y, D

4.2.2 Extending the graph

To extend the graph, we first generate the list of dynamic graphs which is passed as input to the algorithm. We follow this process

We first declare the dynamic graphs, G_i and G_j, from the graph list. Here, i and j represent the moments the graphs were sampled, with $i < j$. We then time the global and local retraining parts separately. In the global retraining process we take as input of the *node2vec* algorithm the resulting graph G_j. For the local retraining we again take as input graph G_j but we define as starting nodes Δ the different nodes of the two graphs G_i, G_j. We also keep track of the nodes traversed from the random walks of the local model. Finally we update the initial embeddings with the embeddings obtained from the new random walks.

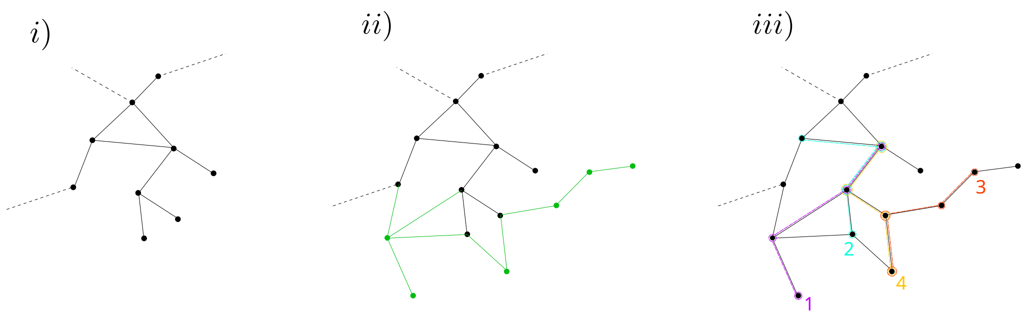


FIGURE 8: Extending the graph. (i) A snapshot of the initial graph. (ii) Nodes are added (green), increasing the graph's size and complexity. (iii) Four random walks are initiated from selected nodes within the newly added portion of the graph.

Algorithm 3: dynamic_extend(G, N, graphs_list)

```
Data: Initial graph $G$, Number of nodes to add $N$, list of graphs $graphs\_list$
Result: Global and local embeddings, times taken for global and local
        embeddings, number of added nodes
$G_i \leftarrow$ first graph in $graphs\_list$;
$G_j \leftarrow$ last graph in $graphs\_list$;
$\Delta = |V_{G_j}| \setminus |V_{G_i}|$;
$model\_i \leftarrow node2vec(G_i)$;
$model\_i.fit()$;
$X_i, y_i, D_i \leftarrow$ emb_group($model\_i$);
global timing
    $model\_global \leftarrow node2vec(G_j)$;
    $model\_global.fit()$;
    $X\_local, y\_global, D\_global \leftarrow$ emb_group($model$);
local timing
    $model\_temp \leftarrow node2vec(G_j, \text{starting\_nodes} = \Delta)$;
    D_temp $\leftarrow$ emb_group($model\_temp$);
    $traversed\_nodes \leftarrow$ nodes from random walks of $model\_temp$;
    $D\_local \leftarrow$ copy of $D_i$;
    update pairs of $D\_local$ with pairs of D_temp for keys in $traversed\_nodes$;
    $X\_local, y\_local \leftarrow$ unpack (node label, embedding) pairs from dictionary
      $D\_local$
return
  $X\_global, y\_global, X\_local, y\_local, total\_global\_time, total\_local\_time, |\Delta|$
```

4.2.3 Pruning the graph

Before we prune the graph, we need to first keep track of the neighbors of the nodes we plan to remove. We introduce an additional variable N_Δ which denotes the random walk neighbors of a nodes set Δ. More specifically, we define an algorithm which traverses a set of nodes Δ of a graph and saves the walk neighbors of Δ that are H steps away

Algorithm 4: get_neighborhood(G, Δ, H)

```
Data: Graph $G$, Nodes set $\Delta$, Maximum number of hops $H$
Result: $H$ steps away neighbor nodes of $\Delta$
$final\_neighbors \leftarrow [\,]$;
for each node in $\Delta$ do
    $current\_neighbors \leftarrow$ neighbors of $node$ in $G$;
    $all\_neighbors \leftarrow current\_neighbors$;
    for $i \leftarrow 1$ to $H$ do
        $new\_neighbors \leftarrow set()$;
        for each neighbor in current_neighbors do
            append $neighbor$ to $new\_neighbors$;
        $current\_neighbors \leftarrow new\_neighbors - all\_neighbors$;
        append $current\_neighbors$ to $all\_neighbors$;
    Remove $node$ from $all\_neighbors$;
    Create $neighborhood\_subgraph$ from $G$ using $all\_neighbors$;
    append nodes of $neighborhood\_subgraph$ to $final\_neighbors$;
return $final\_neighbors$;
```

For the pruning process, we follow a similar approach as in the extension process. We first declare the dynamic graphs, G_i and G_j, with $i > j$. For clarity, we refer to these graphs as *G_updated* and *G_pruned*, respectively. We then time the global and local retraining parts separately. In the global training we take as input of the *node2vec* algorithm the pruned graph *G_pruned.* For the local retraining we first keep track of the different nodes Δ of the pruned and updated (initial) graph. We then obtain the neighbourhood N_Δ of the nodes Δ found in the initial graph. Afterwards, we generate a *node2vec* model and embeddings dictionary *D_pruned* taking as input the pruned graph and as starting nodes the neighbors N_Δ. We create an embeddings dictionary *D_mod* as a copy of the initial embeddings but filtered to the nodes of N_Δ. Finally, we update the embeddings dictionary *D_mod* with the embeddings from *D_pruned* and unpack the local embeddings.

Algorithm 5: dynamic_prune(G, R, graphs_list)

```
Data: Initial graph $G$, Number of nodes to prune $R$, list of graphs $graphs\_list$
Result: Global and local embeddings, times taken for global and local retraining,
        number of removed nodes
$G\_pruned \leftarrow$ first graph in $graphs\_list$;
$G\_updated \leftarrow$ last graph in $graphs\_list$;
$\Delta = |V_{G\_updated}| \setminus |V_{G\_pruned}|$;
$model\_initial \leftarrow node2vec(G)$;
$model\_initial.fit()$;
$X\_initial, y\_initial, D\_initial \leftarrow$ emb_group($model\_initial$);
global timing
    $model\_updated \leftarrow node2vec(G\_pruned)$;
    $model\_updated.fit()$;
    $X\_global, y\_global, D\_global \leftarrow$ emb_group($model\_updated$);
local timing
    $N_\Delta \leftarrow \textbf{get\_neighborhood}(G\_initial, \Delta, 1)$;
    $model\_pruned \leftarrow node2vec(G\_pruned$, starting_nodes $= N_\Delta)$;
    $X\_pruned, y\_pruned, D\_pruned \leftarrow$ emb_group($model\_pruned$);
    $D\_mod \leftarrow$ copy of $D\_initial$ filtered to different nodes $N_\Delta$;
    update pairs of $D\_mod$ with pairs of $D\_pruned$;
    $X\_local, y\_local \leftarrow$ unpack (node label, embedding) pairs from dictionary
      $D\_mod$ ;
return
  $X\_global, y\_global, X\_local, y\_local, total\_global\_time, total\_local\_time, |\Delta|$
```

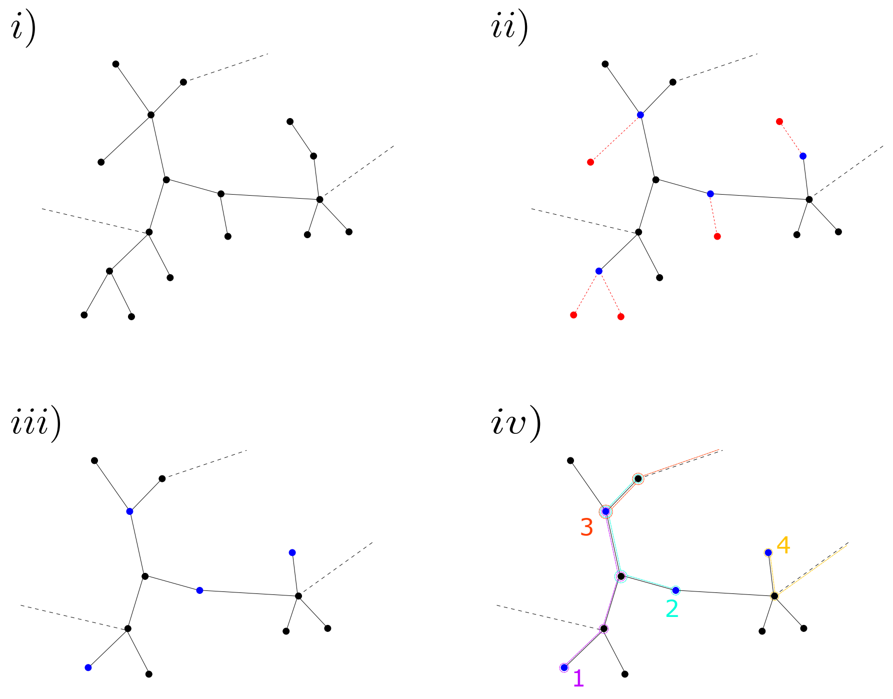


FIGURE 9: Pruning the graph. (i) A snapshot of the original graph. (ii) Nodes are selected (red) such that the graph remains connected after they are removed. (iii) The resulting pruned graph with the removed nodes' neighbors (blue). (iv) Four random walks initiated from the neighbouring nodes.

5 Results

We are evaluating our implementation on the task of node classification. Before comparing the embeddings in terms of accuracy and training time, we first present some experimental results about our implementation.

5.1 Experimental Analysis and Modifications

Ideally, node2vec samples from the distribution of $P(v)$ (see chapter Chapter 2.2.2). However, eliorc's implementation relies on node co-occurrence frequencies from the generated walks instead of the degree distribution, similar to how word2vec uses word frequencies in text data.

It has been proven for DeepWalk (a special case of node2vec where $p = q$) that if the degree distribution of a connected graph follows a power law, the frequency with which vertices appear in the random walks also follows a power-law distribution [Perozzi et al., 2014]. In graph theory, a power law distribution describes a network where a few nodes have many connections, while most nodes have few. It also has a heavy tail, meaning a few nodes have an exceptionally high number of connections. This empirically justifies using frequencies instead of degrees for negative sampling. However, this theoretical basis is only proven for DeepWalk, which uses standard random walks. For the generalized version of node2vec, especially for short random walks, this lemma does not necessarily hold true. Furthermore, *node2vec* relies on biased random walks (as defined in Chapter 2.2.3).

To empirically analyze if sampling from node frequencies is similar to sampling from the degree distribution, we have included graphs of the degree and frequency distributions (see Figures 10, 11). A theoretical analysis and proof would be outside the scope of this project.

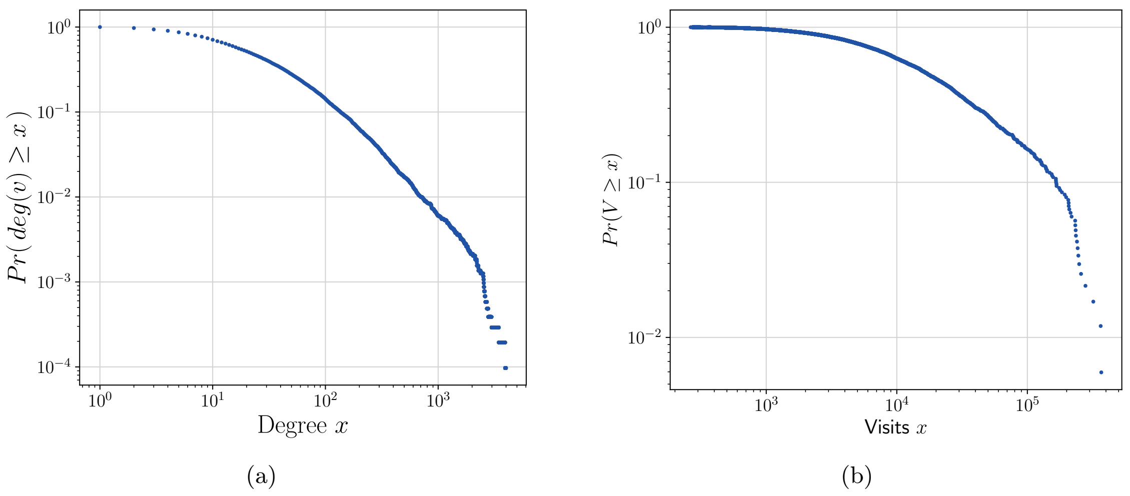


FIGURE 10: Degree (a) and frequency (b) log-log plot. In plot (a) the x-axis represents the degree x of a node v while the y-axis represents the probability that the degree is larger or equal than x. In plot (b) the x-axis represents the visits x of a random walk while the y-axis represents the probability that a node has visits larger or equal than x.

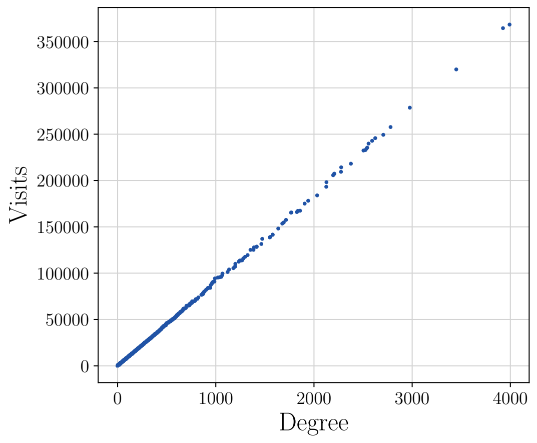


FIGURE 11: Correlation plot of Degree-Frequency.

In Figure 10, both the degree distribution plot (a) and the frequency plot (b) follow a similar distribution. More specifically, both plots exhibit a heavy-tailed distribution. This indicates while most nodes have a relatively low degree, there are a few nodes with a very high degree.

In Figure 11, it appears that degree and frequency are directly correlated, which is indicated by the linear relationship observed in the scatter plot. Specifically, as the degree of a node increases, the number of visits also increases proportionally. This correlation suggests that nodes with higher connectivity (higher degree) are visited more frequently during the random walks. This direct correlation can be explained by the nature of random walks as they more likely to visit highly connected nodes due to their increased number of edges.

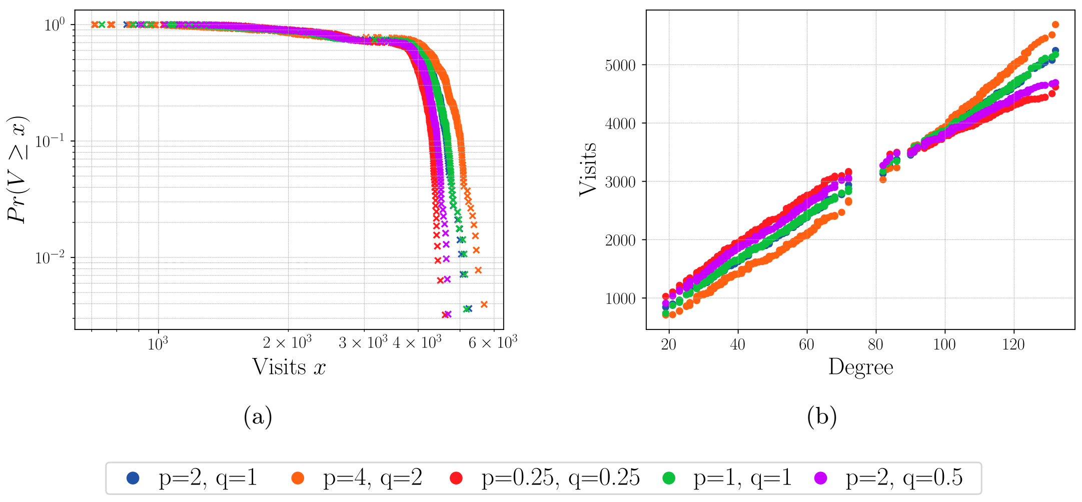


FIGURE 12: Plots

We also plot the frequency and correlation plots for multiple (p, q) configurations. We again observe that they all follow a similar distribution (Figure 12a) and have direct correlation (Figure 12b).

5.2 Dataset F-scores

Metric	Retraining	10%	20%	30%	40%	50%	60%	70%	80%	90%
Micro-F1	Global	0.7178	0.7535	0.7581	0.7730	0.7940	0.8082	0.8173	0.8167	0.8371
	Local	0.7350	0.7628	0.7758	0.7846	0.8058	0.8173	0.8253	0.8229	0.8487
Macro-F1	Global	0.7511	0.7650	0.7722	0.7951	0.7982	0.8195	0.8217	0.8454	0.8457
	Local	0.7596	0.7766	0.7827	0.8043	0.8058	0.8201	0.8229	0.8450	0.8524

TABLE 2: Performance metrics for Cora dataset for $(d, r, l, p, q) = (128, 40, 80, 0.25, 1)$.

In Table 2, we gather the F1-scores for a specific dataset. We define the column "Retraining" for differentiating between *global* retraining (taking as input all the graph's nodes) and *local* retraining (taking as input only the affected area of the graph). We also contain "training size percentage" columns ranging from 10% to 90%, representing the micro/macro F1-scores for different training/test ratios. E.g. 10% means training size $= 0.9 *$ dataset size.

We create this table for every choice of *node2vec* hyperparameter list (d, r, l, p, q) and type of dynamic update (extending/pruning) for each dataset. From now on, we define $(d, r, l, p, q) = (128, 40, 80, 0.25, 1)$ as the default *node2vec* hyperparameter list (see Chapter 5.4 for explanation).

As expected, the scores rise as the training size increases. However, for every dataset the scores can differ. For example, the BlogCatalog dataset has much lower F1-scores compared to the other datasets (see Figure 13).

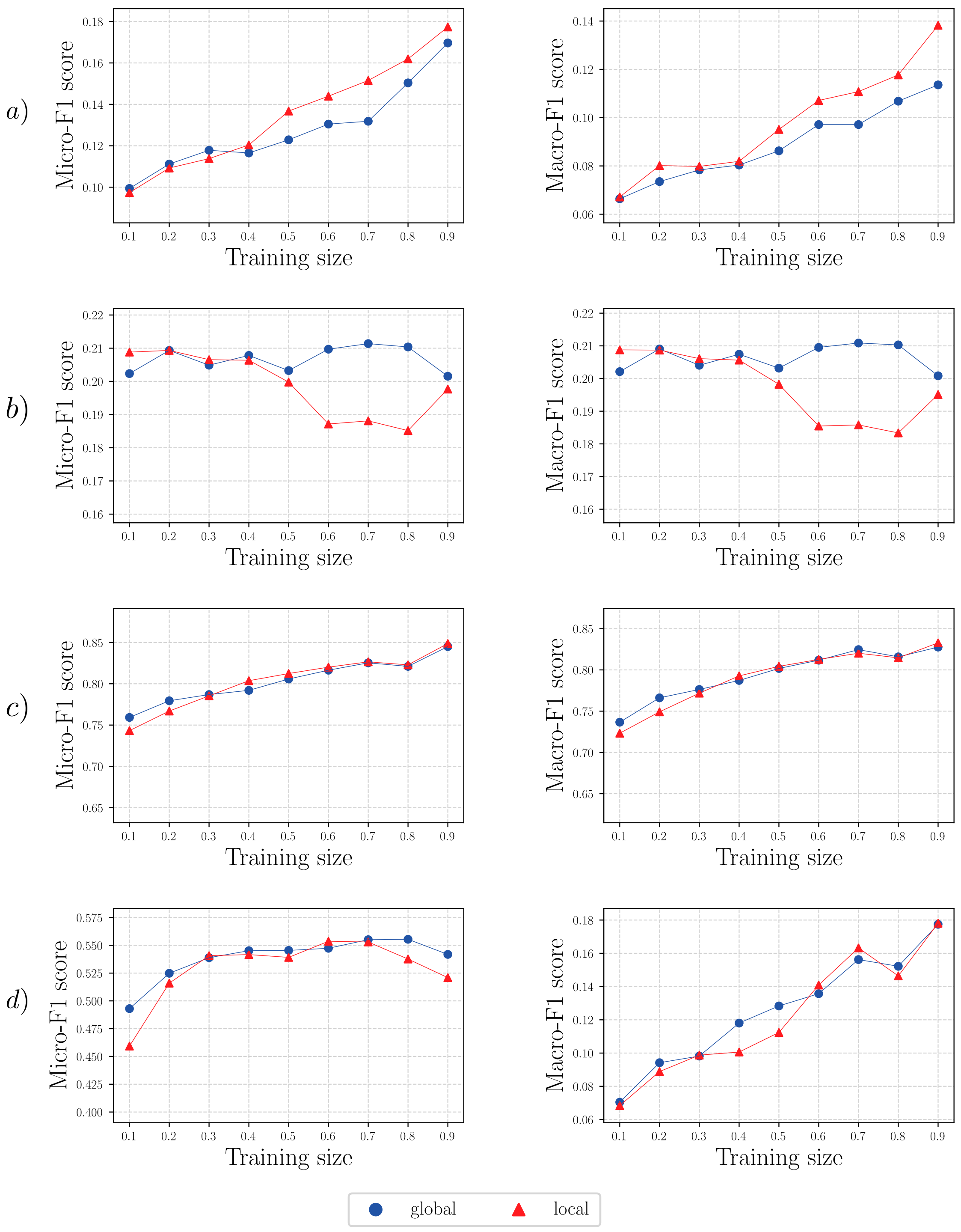


FIGURE 13: Comparison of global and local retraining for (a) PPI, (b) BlogCatalog, (c) Cora and (d) Wikipedia datasets using F1-scores. Blue circles represent global retraining, while red triangles represent local retraining.

The scores for each dataset are found in Figure 13. In the PPI, Cora and Wikipedia datasets, micro-F1 scores of both global and local retraining show an increasing trend with training size, with local retraining (red triangles) slightly outperforming global retraining (blue circles) at larger sizes. This also appears to be true for the macro-F1 scores. The superior performance of local retraining might be attributed to its ability to capture local structures more effectively, although the differences are minimal.

In the case of the BlogCatalog, it seems that after 50% of the training size, the global scores seem to outperform the local ones. Nevertheless, the differences between the two strategies remain very small.

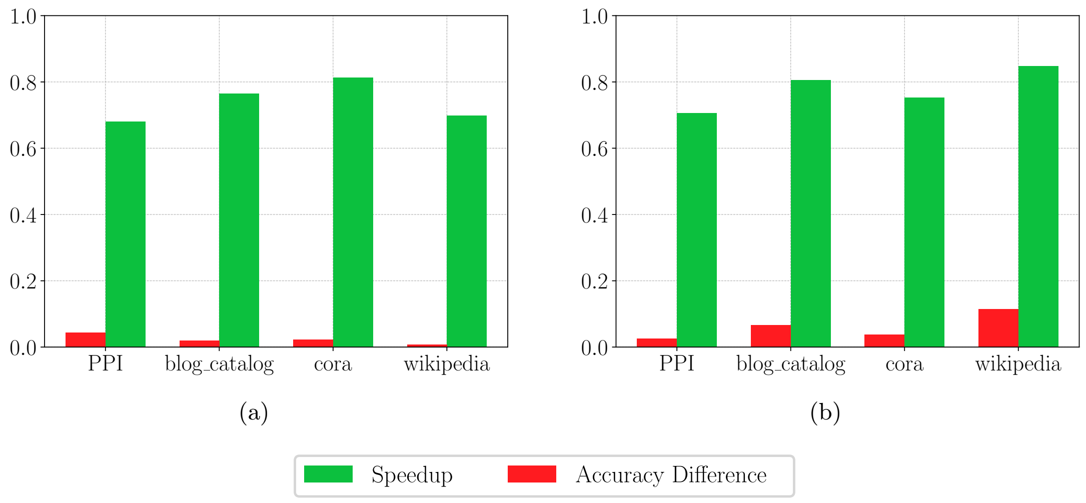
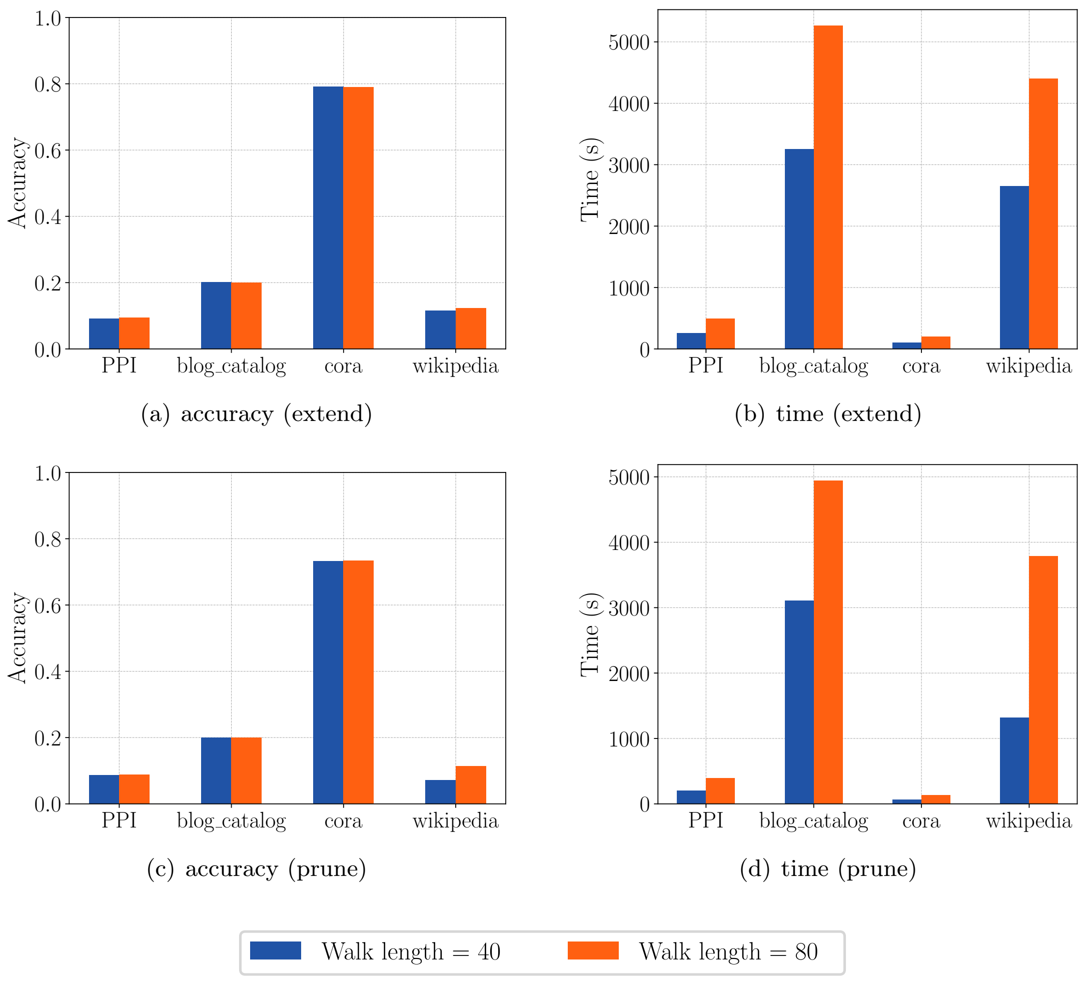


Figure 14: Bar charts comparing dataset time speedup and accuracy difference for (a) extend and (b) prune methods across datasets. Green bars represent speedup, while red bars represent accuracy difference.

Another important measure of performance is presented in Figure 14. The figure averages the macro F1-scores for each dynamic update process and for each training percentage using the default hyperparameter list. The red bar represents the accuracy difference between retraining locally and globally. It seems that the difference is miniscule. In contrast, the difference in time speedup (green bar) is substantial, reaching at least 70% in speedup gains for both types of dynamic update. More specifically, when extending the graph the cora dataset reaches an 80% gain while keeping an accuracy loss of less than 5%. In the case of pruning, the wikipedia dataset seems to score the highest speedup gain among the datasets, reaching almost 85%. However, it also has the highest accuracy loss among all datasets passing 10%, even though it is still relatively small.

5.3 Random Walk Length

Another important comparison can be made on the length of the random walks. More specifically, we perform experiments for a walk length = 40 and walk length = 80 for both types of dynamic update. We take the average results of all hyperparameter sets (see Chapter 5.4 for hyperparameter details).



(a) accuracy (extend)

(b) time (extend)

(c) accuracy (prune)

(d) time (prune)

FIGURE 15: Comparison of walk lengths in terms of accuracy and time across datasets. Blue bars represent a walk length of 40, while orange bars represent a walk length of 80.

From Figure 15 we observe that in the case of extending, both random walk lengths seem to offer almost identical accuracy. In comparison, choosing walk length = 40 seems to offer a dramatic time difference. For every dataset, the amount of time used to retrain the model is doubled when the walk length = 80 on average. For BlogCatalog a time difference of almost 40% is observed. In the wikipedia dataset a difference of is found.

In the case of pruning, a similar behaviour is observed as in extending. The main difference lies in the wikipedia dataset. In terms of accuracy, having a longer walk length seems to give almost double the accuracy. In terms of time, both walk lengths offer a higher speedup, especially in the case of the shorter walk length which is more than 60% faster than the longer random walk.

5.4 Hyperparameter Configurations

We perform grid search on the hyperparameter space of the node2vec algorithm. Grid search systematically generates all possible combinations of these values, forming a Cartesian product of hyperparameter sets [Goodfellow I., 2016].

We define the hyperparameter space as follows

$$(d, r, l, p, q) \in \{64, 128\} \times \{8, 10, 20, 40, 80\} \times \{8, 80, 160, 240\} \times \{0.25, 0.5, 1, 2, 4\} \times \{0.25, 0.5, 1, 2, 4\}$$

where

- d is the vector dimension
- r is the number of random walks
- l is the random walk length
- p the return parameter
- q the in-out parameter

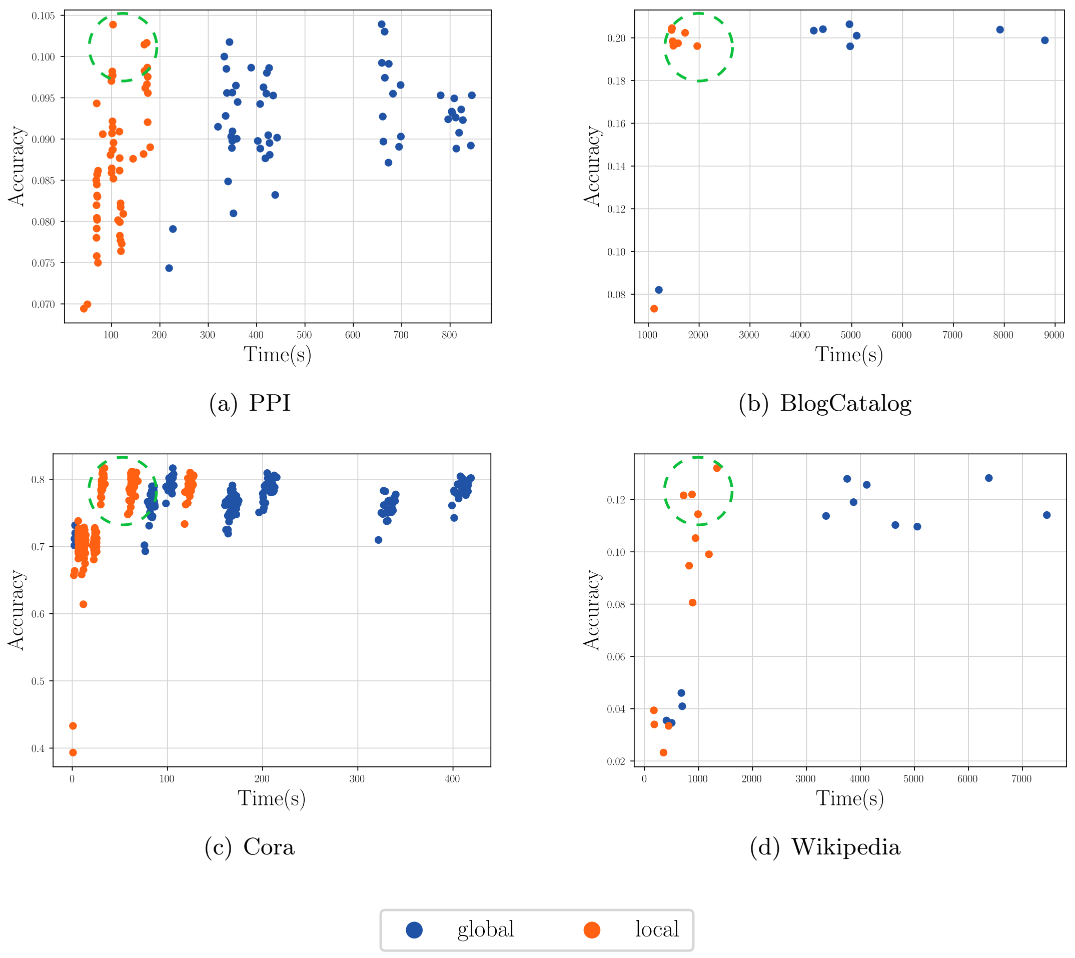


Figure 16: Scatter plots comparing accuracy and time in terms of global and local retraining across datasets. Blue dots represent global retraining and orange dots represent local retraining. Green dotted circles represent the hyperparameter intersection.

In Figure 16, we observe that local retraining performs much better than global in terms of time while keeping a high accuracy. This plot takes into account the whole hyperparameter set where each dot corresponds to a different (d, r, l, p, q) configuration.

We take the intersection of the parameters that maximise accuracy for each dataset in terms of local retraining. From our analysis, it seems that $(128, 40, 40, 0.25, 1)$ is the optimal configuration. The relative low value of the return parameter $p = 0.25$ makes the walks more probable to backtrack and stay close to the starting node. Additionally, as discussed in Chapter 5.3, selecting $l = 40$ yields precise results while significantly reducing computation time.

We also choose 512 as the number of nodes when generating dynamic graphs. This choice ensures that the implementation remains computationally feasible while still providing a representative baseline for each network.

5.5 Node Removal Strategies

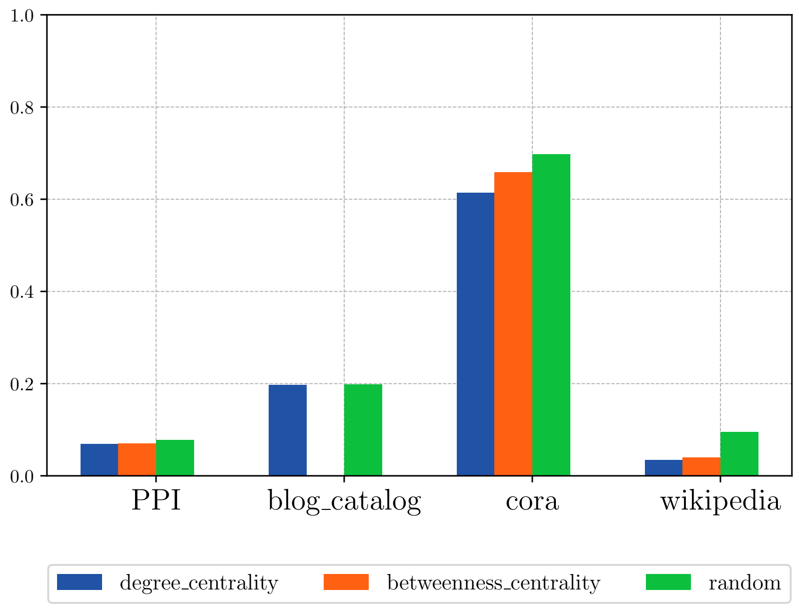


Figure 17: Bar charts comparing accuracy for different node removal processes across datasets. Blue bars represent node removal via degree centrality, orange bars via betweenness centrality and green bars via removing nodes randomly.

In Figure 17, we observe that choosing a random node to remove when generating the dynamic graphs is always more accurate than removing nodes based on centrality measures. However, the differences are minimal. Specifically, the degree centrality method consistently results in the lowest accuracy across all datasets, indicating that removing nodes with the highest number of connections disrupts the network structure more than other methods.

Another interesting point is the time complexity of the different removal processes. As mentioned in Chapter 4.2.1, betweenness centrality as a node removal strategy is computationally expensive. Calculating betweenness centrality for all nodes in a graph has a time complexity of $O(n^3)$ in the worst case, where n is the number of nodes. Degree centrality has a time complexity of $O(n+m)$ where n is the number of nodes and m the number of edges, as it initialises a degree count for each node ($O(n)$) and iterates through all edges, where each edge increments the degree count of the node it connects ($O(m)$). In comparison, selecting a random node to remove takes $O(1)$.

Given its superior accuracy and lower computational complexity, random node removal emerges as the best overall strategy.

6 Conclusion and Discussion

The main goal of this research was to optimize node2vec for dynamic graphs and evaluate its performance in terms of time and accuracy. Reflecting on our first two research questions about making node2vec more efficient in dynamic graphs through approximations, we demonstrated that local training, rather than retraining the entire updated graph, yields similar results. Although this approach results in a relatively small loss in accuracy, it provides a significant speedup. Regarding the third research question on how the selection of a node for removal impacts the model's accuracy, our findings indicate that different removal methods lead to varying accuracy outcomes, though these differences were relatively minor. Notably, random node removal emerged as the most accurate and efficient strategy among the methods we tested.

It is important to note the limitations of our study. First of all, there are imbalances and differences in our datasets, particularly with the much larger BlogCatalog dataset. We removed the same number of nodes from every dataset. While this number was sufficient for most datasets, representing a significant portion of each graph, BlogCatalog might have benefited from removing a larger number of nodes. Additionally, a greater variety in the number of nodes removed could have been more beneficial. However, the main constraint was the computational complexity of the random walks, as the algorithm needs to traverse every node, making the time scale by the size of the graph and the node set. We were unable to perform extensive hyperparameter tuning on the BlogCatalog dataset, unlike the more comprehensive testing conducted on other datasets due to the complexity of the random walks and the size of the dataset.

Another point to consider is the different structure of each graph. It is highly probable that graphs with different statistics (e.g., number of nodes/edges, number of connected components, clustering coefficient, etc.) would yield different results. Additional experiments could be done in more datasets to recognise possible patterns when removing specific nodes. The analysis could also be coupled with graphs statistics (e.g. clustering coefficient, degree statistics etc.) to understand the effect of removing specific nodes.

Additionally, exploring new methods for extending and pruning the graph could yield valuable insights. For instance, when extending the graph, it would be intriguing to examine the impact of attaching another graph to the existing one and analyze how the embeddings change based on the structure of the new graph (e.g. if the attaching graph has the structure of a tree graph, a path graph etc.).

A Hardware Implementation

Running Environment. The experiments are conducted on a single CPU-only Linux server with AMD EPYC 7713P 64-Core Processor and 128G RAM. The number of parallel workers is 64, equal to the number of cores.

References

Mohammadreza Armandpour, Patrick Ding, Jianhua Huang, and Xia Hu. Robust negative sampling for network embedding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):3191–3198, July 2019. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v33i01.33013191.

Melania Berbatova. *Using Structured Information from Tags for Book Recommendations.* PhD thesis, 03 2020.

Christopher M. Bishop. *Pattern recognition and machine learning.* Information science and statistics. Springer, New York, 2006. ISBN 978-0-387-31073-2.

Ulrik Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social networks*, 30(2):136–145, 2008.

Bobby-Joe Breitkreutz, Chris Stark, Teresa Reguly, Lorrie Boucher, Ashton Breitkreutz, Michael Livstone, Rose Oughtred, Daniel H Lackner, Jürg Bähler, Valerie Wood, et al. The biogrid interaction database: 2008 update. *Nucleic acids research*, 36(suppl_1): D637–D640, 2007.

Hugo Caselles-Dupré, Florian Lesaint, and Jimena Royo-Letelier. Word2vec applied to recommendation: hyperparameters matter. In *Proceedings of the 12th ACM Conference on Recommender Systems*, page 352–356, Vancouver British Columbia Canada, September 2018. ACM. ISBN 978-1-4503-5901-6. doi: 10.1145/3240323.3240377. URL `https://dl.acm.org/doi/10.1145/3240323.3240377`.

eliorc. Node2vec. `https://github.com/eliorc/node2vec`, 2022. Accessed: 2024-05.

Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40 (1):35, March 1977. ISSN 00380431. doi: 10.2307/3033543.

Gensim. Word2vec. `https://radimrehurek.com/gensim/models/word2vec.html`, 2024. Accessed: 2024-05.

Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. (arXiv:1402.3722), February 2014. URL `http://arxiv.org/abs/1402.3722`. arXiv:1402.3722 [cs, stat].

Courville A. Goodfellow I., Bengio Y. *Deep learning.* The MIT Press, 2016.

Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.

Aditi Juneja. nx-parallel. `https://github.com/networkx/nx-parallel/tree/main`, 2024. Accessed: 2024-07.

Nobuhiro Kaji and Hayato Kobayashi. Incremental skip-gram model with negative sampling. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 363–371, Copenhagen, Denmark, September 2017. Association for Computational Linguistics. doi: 10.18653/v1/D17-1037. URL `https://aclanthology.org/D17-1037`.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Matt Mahoney. Large text compression benchmark. http://www.mattmahoney.net/dc/textdata, 2011.

Christopher D Manning, Prabhakar Raghavan, and Hinrich Schutze. Introduction to information retrieval.

Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3: 127–163, 2000.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.

Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 5363–5370, 2020.

Hao Peng, Jianxin Li, Hao Yan, Qiran Gong, Senzhang Wang, Lin Liu, Lihong Wang, and Xiang Ren. Dynamic network embedding via incremental skip-gram with negative sampling. *Science China Information Sciences*, 63:1–19, 2020.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

Guillaume Salha, Romain Hennequin, and Michalis Vazirgiannis. Simple and effective graph autoencoders with one-hop linear models. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14–18, 2020, Proceedings, Part I*, pages 319–334. Springer, 2021.

Abdel Aziz Taha and Allan Hanbury. Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. *BMC medical imaging*, 15:1–28, 2015.

Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Dyrep: Learning representations over dynamic graphs. *International Conference on Learning Representations (ICLR)*, 2019. URL https://openreview.net/references/pdf?id=Hy8cqH0rE.

Zhen Yang, Ming Ding, Chang Zhou, Hongxia Yang, Jingren Zhou, and Jie Tang. Understanding negative sampling in graph representation learning. page 1666–1676, August 2020. doi: 10.1145/3394486.3403218. URL https://dl.acm.org/doi/10.1145/3394486.3403218.

R. Zafarani and H. Liu. Social computing data repository at ASU, 2009. URL http://socialcomputing.asu.edu.