REGPATTERN2VEC KNOWLEDGE GRAPH EMBEDDING

by

SAKETH REDDY JAMMULA

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

RegPattern2Vec [1] is a novel algorithm used for knowledge graph embedding. It effectively samples a large Knowledge Graph to learn node embeddings, while capturing the semantic relationships between the nodes of graph. This thesis proposes an implementation of the RegPattern2Vec algorithm as a custom plugin in the Neo4j graph database. This plugin generates random walks, which can then be used to generate vector embeddings. The plugin accepts a regular expression transformed Finite Automata as one of the inputs, and generates random walks based on this Finite Automata as output, all of which is implemented in Java. The generated random walks through this procedure effectively capture the semantic relationship between the graph nodes with minimum prior knowledge and human involvement. The plugin is then successfully tested on a subset citation network dataset and Olympic Results dataset and the results are documented.

INDEX WORDS: RegPattern2Vec, knowledge graph embedding, Neo4j, random walks.

REGPATTERN2VEC KNOWLEDGE GRAPH EMBEDDING

by

SAKETH REDDY JAMMULA

B.Tech., Vardhaman College of Engineering, 2018

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2023

REGPATTERN2VEC KNOWLEDGE GRAPH EMBEDDING

by

SAKETH REDDY JAMMULA

Major Professor:	Krzysztof J. Kochut
Committee:	Lakshmish Ramaswamy Khaled Rasheed

Electronic Version Approved:

Ron Walcott
Vice Provost for Graduate Education and Dean of the Graduate School
The University of Georgia
August 2023

To my parents, my family, my teachers, and my friends

ACKNOWLEDGEMENTS

I would like to thank all my committee members for their time and suggestions throughout my research. A special thanks to my major advisor Prof. Krzysztof J. Kochut who shed a light on my path with his great guidance, support, and motivation throughout my study. I am extremely grateful to my family and friends who have helped me to keep moving and achieve my goals. I would also like to thank everyone at the *College of Computing* and the *University of Georgia* who have guided me at every step of my masters' journey.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

In a world with exponentially increasing data, Knowledge Graphs (KG) are essential as they provide a structured and organized way to represent, store, and analyze complex and interrelated information. Knowledge Graphs involve storing data in a graph structure, where entities of different types are represented as nodes, and the relationships between these entities are represented as edges connecting the nodes. Furthermore, by incorporating metadata such as labels, descriptions, and additional attributes, Knowledge Graphs offer more context and meaning to the data. A formal graph-based definition of Knowledge Graph is provided in Definition 1.1. While Knowledge Graphs provide a structured representation of data and capture the relationships between entities, graph embeddings provide vector representation of nodes or edges in a graph. These embeddings are numerical representations that capture the structural and semantic information of the graph. The relationship between these vectors reflects the relationship between the graph elements. Due to computational efficiency, feature extraction, generalization, scalability and integration with machine learning models, graph embeddings enable more efficient and effective analysis of graph data. A formal definition of graph embeddings is provided in Definition 1.2.

Definition 1.1 (Knowledge Graph). A Knowledge Graph G: (V, E) whose nodes $v_i \in V$ are entities and edges $e_i \in E$ are relations connecting the entities. For each node in V, we have a type mapping function Φ: $V \rightarrow T$, where T denotes a node type set, and edges have an associated type mapping

function $\Phi: E \rightarrow R$ where R denotes a relation type set. Thus, a triple (v_i, e, v_j) forms an edge which implies the relation e between two nodes v_i and v_j. (Keshavarzi et al., 2021)

Definition 1.2 (Graph embedding). A graph embedding determines a fixed length vector representation for each entity (usually nodes) in our graph. These embeddings are a lower dimensional representation of the graph and preserve the graph's topology [2].

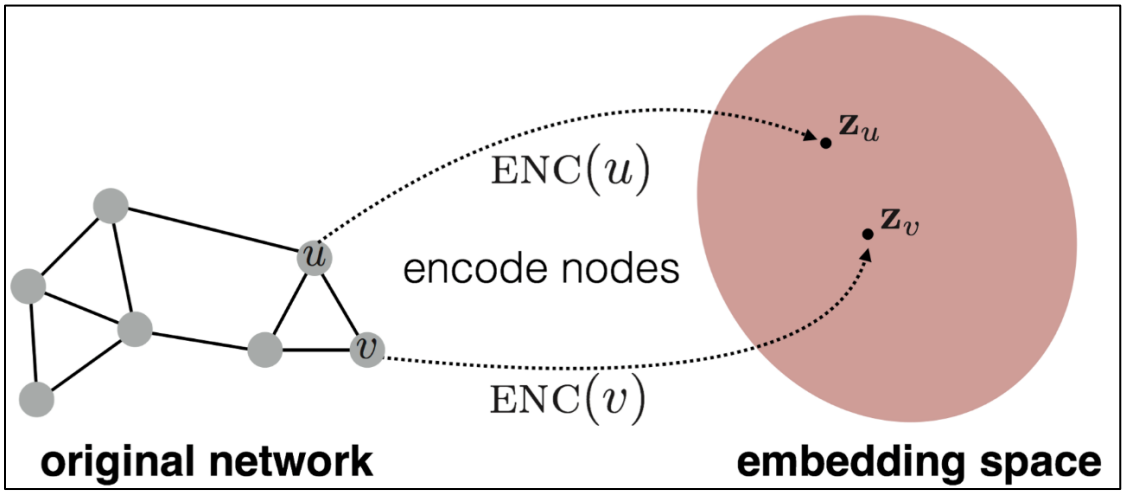


Figure 1. Graph embedding maps graph data into feature vectors [3]

Knowledge Graphs (KG) serve as a valuable resource for Machine Learning algorithms, and find applications in domains such as healthcare, social networks, recommendation systems, bioinformatics, and more, facilitating knowledge discovery. In healthcare, KGs can be used to represent patient data, medical information, enabling better treatment. In social networks, KGs can be used to represent user data, their interests, enabling better content recommendation. In recommendation systems, KGs can be used to represent user behavior, enabling to build personalized products and services by the companies. Machine Learning (ML) on Knowledge Graphs can be used for link prediction, where Machine Learning models can be used to predict missing or future relationships between nodes in a Knowledge Graph. ML algorithms leverage the

graph structure, the associated node features and help in node classification, which is important in fraud detection and sentiment analysis. One more popular application of ML on Knowledge Graphs is the ability to map nodes of the graph as equivalent low-dimensional vectors, capturing semantic and structural information of the graph nodes. These vectors are graph embeddings, and this work focuses on using these embeddings. Graph embeddings enable downstream Machine Learning algorithms to operate directly on the graph data effectively and efficiently.

Neo4j is a graph database management system used to work with Knowledge Graphs. It is very good for querying interrelated information. It provides powerful graph traversal capabilities, making it easy to navigate graph structures and extract insights from the data. Neo4j is highly scalable for managing Knowledge Graphs that has large amount of interrelated information. Neo4j has its own query language called Cypher, designed for querying, and manipulating graph data in Neo4j. As these graph database tools have become more accessible, the demand for graph algorithms that can operate directly on graph databases has increased. As we now understand, data is becoming increasingly connected and it has become increasingly important to understand its interdependencies. Graph algorithms help identify patterns in the highly connected datasets and help understand the structural information of the data. However, as the scale and complexity of graph data grows, there arises a need for a different approach to analyze and represent this data. This is where graph embeddings come into play capturing contextual information of nodes, the relationships between them, and even the attributes associated with them, which can effectively be used for downstream Machine Learning tasks. The Neo4j Graph Data Science library [4] contains embedding algorithms that can be used for Machine Learning. These algorithms include Node2Vec [5], GraphSAGE [6], Fast Random Projection [7], HashGNN [8].

Some prominent graph embedding algorithms such as Node2Vec, DeepWalk [9], metapath2vec [10] have been proposed and implemented in the literature, however, all these algorithms come up with their own set of limitations, which are discussed in detail in Chapter 2 of this thesis. To overcome one of those limitations, a new graph embedding algorithm has been implemented which is RegPattern2Vec (Keshavarzi et al., 2021). This thesis proposes an implementation of this RegPattern2Vec algorithm as a custom plugin in the Neo4j graph database. This plugin generates random walks, which can then be used to generate vector embeddings. The plugin is then successfully tested on a citation network dataset and Olympic Results dataset and the results are documented.

This thesis is organized into 5 chapters. Chapter 1 provides an overview of the ecosystem of the research topic and summarizes the work done in this thesis. Chapter 2 reviews the related work in graph embedding algorithms, discusses the distinction between homogeneous graphs, heterogenous graphs, and then introduces the need for regular patterns on Knowledge Graphs. Chapter 3 discusses graph embeddings based on random walks, then further discusses Node2Vec, DeepWalk, metapath2vec embedding algorithms generating graph embeddings based on random walks, and then introduce the RegPattern2Vec algorithm embedding generation. Chapter 4 discusses the overall methodology underlying the proposed approach, implementation and explains in detail the methods used. Chapter 5 presents the dataset, experimental results of the proposed approach, and overall pipeline of the system. We conclude this thesis and discuss future work in Chapter 6.

CHAPTER 2

RELATED WORK

In recent years, graph embedding algorithms have emerged as powerful techniques for learning low-dimensional representations of nodes in networks. Popular embedding algorithms such as DeepWalk, Node2Vec, metapath2vec have leveraged random walks on networks to capture structural information and generate effective graph embeddings. Machine Learning models today rely on numerical input rather than discrete structures or symbols. There arises a need to convert the input in the form of a machine understandable numerical representation. Embeddings have become a representation of choice serving as input to these machine learning models. Word embeddings are vector representation of words that capture semantic and contextual information. These are numerical representations that encode the meaning of words in a high-dimensional space, where words are located close to each other. Word2Vec [11] is one of the popular word embedding algorithms used to predict neighboring words, given a target word, or vice versa.

Graph embeddings, similar to word embeddings, aim to represent nodes as low-dimensional vectors. Instead of words, graph embeddings capture the structural and relational information of nodes and edges within a graph. This representation facilitates the application of machine learning algorithms and enables various tasks such as link prediction, node classification, and recommendation systems. Node2Vec, DeepWalk and metapath2vec are random walk-based graph embedding algorithms. The approach presented in this thesis is motivated from these random walk-based embedding approaches. Definition 1.3 provides a clear explanation of random walks.

Definition 1.3 (Random walk). A random walk in a Knowledge Graph can be formally defined as a stochastic process that traverses the graph by moving from one node to another based on randomly chosen edges. At each step of the random walk, the next node to visit is determined by randomly selecting an outgoing edge from the current node. The selection can be uniform or biased, depending on the specific random walk algorithm. [12]

The proposed method in this thesis focuses on heterogeneity of nodes and edges in graphs. In this section, we discuss the distinction between homogeneous graphs, heterogenous graphs, and introduce the need for regular patterns on Knowledge Graphs. In Chapter 3, we discuss in detail the current existing embedding algorithms (DeepWalk, Node2Vec, metapath2vec), and then proceed to explain about the RegPattern2Vec algorithm (Keshavarzi et al., 2021).

Homogeneous graphs. A homogeneous graph in the context of Knowledge Graphs refers to a graph where all nodes and edges belong to the same type. In other words, there is a uniformity in the node and edge types throughout the graph, and there is no distinction between different entities or relationships. DeepWalk and Node2Vec algorithms primarily work on homogenous graphs.

Heterogeneous graphs. A heterogeneous graph in the context of Knowledge Graphs refers to a graph where nodes and edges can belong to different types, representing diverse entities and relationships. Unlike homogeneous graphs, heterogeneous graphs incorporate varying node and edge types to capture the rich semantics and complex relationships within the graph.

Metapath2vec algorithm works on heterogeneous graphs. With this type of graphs, where there are multiple type of nodes and edges, metapath2vec uses a fixed path of node types called a meta-path (Dong et al., 2017). Domain experts are needed to choose the meta-paths and the choice of meta-paths also poses challenges. To overcome this, Keshavarzi et al. (2021) introduced the RegPattern2Vec algorithm where a regular expression guides the random walks to sample

sequences of nodes in a more efficient way. The proposed method in this thesis is the Neo4j implementation of the RegPattern2Vec algorithm.

CHAPTER 3

GRAPH EMBEDDINGS BASED ON RANDOM WALKS

Graph embedding based on random walks is an approach for generating node representations in a graph. The process of generating node representations involves random walk generation, consideration of walk length and number of walks, usage of skip-gram or similar models, generating the embeddings (low-dimensional vectors). By employing random-walk based graph embedding, the graph structure can be effectively transformed into a format compatible with classical algorithms, enabling the application of well-established techniques for analyzing and solving graph-based problems.

3.1 Random walk generation

In the context of graph embeddings, random walk generation involves traversing a graph from a specific node and traversing to neighboring nodes in a random manner. It begins with selecting a starting node in the Knowledge Graph at random. Once a starting node is chosen, it acts as the current node which represents the current position in the graph traversal. At each step of the random walk, the next node to visit is determined by randomly selecting the neighboring nodes of the current node. A node is considered a neighboring node if it is connected directly to the current node by an edge. The selection of the next node depends on the transition probabilities which affects the likelihood of a particular node selection. This process is repeated for a specific number of times until the specific condition is met.

3.2 Walk Length and Number of walks per node.

Two important parameters to consider in the process of generating embeddings based on random walks are walk length and number of walks per node. The choice of walk length and number of walks per node depends on the task at hand.

Walk Length. The walk length refers to the number of steps or nodes visited in each random walk. It determines how far a random walk will explore the graph from the starting node. A longer walk length allows for capturing more local context and capturing relationships between nodes that are farther apart. Shorter walk lengths, on the other hand, focus on immediate neighborhood exploration. The length of the random walk determines how much information is captured about the local neighborhood of each node.

Number of walks per node. This parameter affects the quality and coverage of the learned embeddings. It refers to how many random walks are generated for each starting node. Performing multiple walks per node captures different neighborhood structures by traversing different possible paths. A higher number of walks per node provides a more comprehensive exploration of the graph, increasing the chances of discovering different paths and capturing diverse structural information. However, it also increases the computational overhead.

3.3 Skip-gram.

The skip-gram model is an efficient method for learning vector representations that capture word relationships. It predicts the context word, given a target word. By training the model to predict the surrounding words, the model learns to capture the semantic and structural relationships between words. This approach has been used in word2vec.

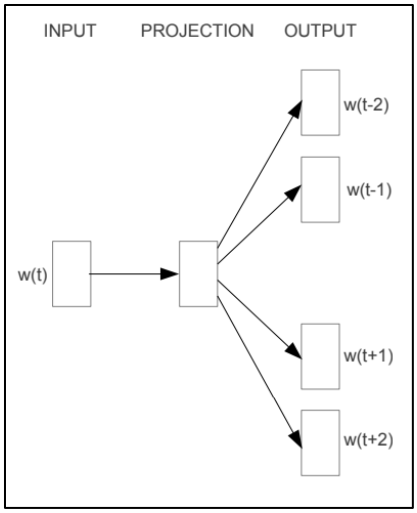


Figure 2. Skip-gram predicts surrounding words given current word (Mikolov et al., 2013)

DeepWalk, node2Vec, metapath2vec algorithms also leverage the skip-gram model as a core component for learning node embeddings from graph data. The objective is to predict the context nodes, given a target node. The skip-gram model is adapted to capture the relationship between nodes in the Knowledge Graph. Instead of words, these algorithms consider the nodes in the graph as the units to be embedded. RegPattern2Vec uses the modified version of the skip-gram model (Dong et al., 2017) and produces the node embeddings. These embeddings capture the similarity of nodes based on their types as required by the regular pattern.

3.4 Embedding generation.

The output of the skip-gram model are the node embeddings, where each node is represented as a low-dimensional vector, capturing the structural and relational information in the Knowledge Graph. These embeddings capture both the local and global structural information of the nodes, as they are learned based on the patterns and relationships observed during the random walk process. These node embeddings can be utilized for a wide range of downstream tasks, such as link prediction, node classification, and recommendation systems.

3.5 DeepWalk, Node2Vec, metapath2vec generation of graph embeddings

DeepWalk. DeepWalk (Perozzi et al., 2014) is an algorithmic framework that treats graph as a collection of sequences and apply techniques inspired by word embeddings. Perozzi et al., (2014) draw an analogy between nodes in a graph and words in a corpus, where nodes that are in close proximity to each other are considered similar. DeepWalk does not explicitly differentiate between the types of nodes or the type of network. It treats all nodes in the Knowledge Graph as equal and focuses on capturing the structural information and neighborhood relationships among nodes. The random walk generation is based on the connectivity patterns in the graph, traversing by following edges and do not considering any node attributes. There arises an exploration-exploitation issue as DeepWalk uses uniform random walk strategy, i.e., each step randomly selects a neighboring node to traverse, thereby does not provide explicit control over the tradeoff between exploring different parts of the graph and exploiting the local neighborhood information effectively. DeepWalk treats all nodes and edges equally and does not consider different types of network structures or node characteristics explicitly. It does not incorporate any mechanisms to differentiate between different node types. DeepWalk does not provide flexibility in controlling the exploration bias during the random walk process. It does not consider higher-order structural information or node attributes that can guide the walk towards specific structures or neighborhoods of interest. To allow better control over certain limitations of DeepWalk algorithm, another algorithm Node2Vec was proposed by Grover and Leskovec (2016).

Node2Vec. Node2Vec (Grover & Leskovec, 2016) is an algorithmic framework that aims to learn continuous feature representations for nodes in networks. It operates by taking a graph as input and extracting a collection of random walks from the graph. These random walks can be seen as sequences of directed words, with each node representing a word. The generated random walks

are then fed into the skip-gram model [13]. By applying the skip-gram model, Node2Vec generates embeddings for each node, providing a compact representation capturing the essential features of the nodes. In contrast to DeepWalk, this algorithm uses biased random walk strategy, that combines breadth-first search and depth-first search strategy, thus providing the control over exploration-exploitation tradeoff. Node2Vec also incorporates two parameters: the return parameter (**p**), the in-out parameter (**q**). The return parameter (**p**) controls the likelihood of returning to the previous node in the random walk. **p** affects the random walk behavior and determines how much the walk explores different parts of the graph. If **p** is high, the random walk is more likely to revisit previous nodes. If **p** is low, the random walk is more likely to move to new nodes. The in-out parameter (**q**) controls the likelihood of moving away from the previous node in the random walk. This parameter allows to control the degree of exploration and localness in the random walk.

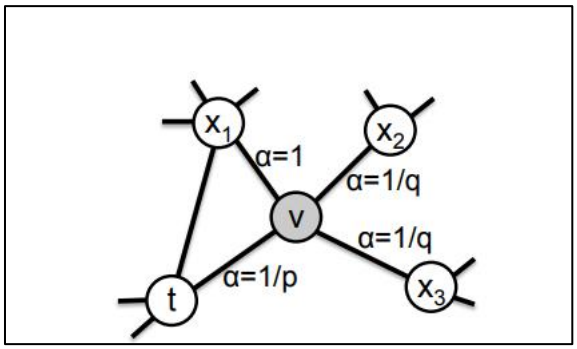


Figure 3. Illustration of the random walk procedure in node2vec. The walk just transitioned from t to v and is now evaluating its next step out of node v (Grover & Leskovec, 2016)

By fine-tuning the **p** and **q** parameters, a balance between exploring different parts of the Knowledge Graph and exploiting the local neighborhood information is attained. These biases allow us to capture both structural information and broader community structure. Despite these

advancements, Node2Vec algorithm still relies on random walks and does not explicitly consider other structural information of the Knowledge Graph. It does not make distinctions on the types of nodes; hence the algorithm deals only with homogenous graphs. These are overcome by the metapath2vec algorithm (Dong et al., 2017).

metapath2vec. metapath2vec (Dong et al., 2017) is an algorithmic framework that uses meta-path-based random walks to learn feature representations for nodes that have different type within the same Knowledge Graph. A meta-path is a path that defines how a random walker should traverse different types of nodes.

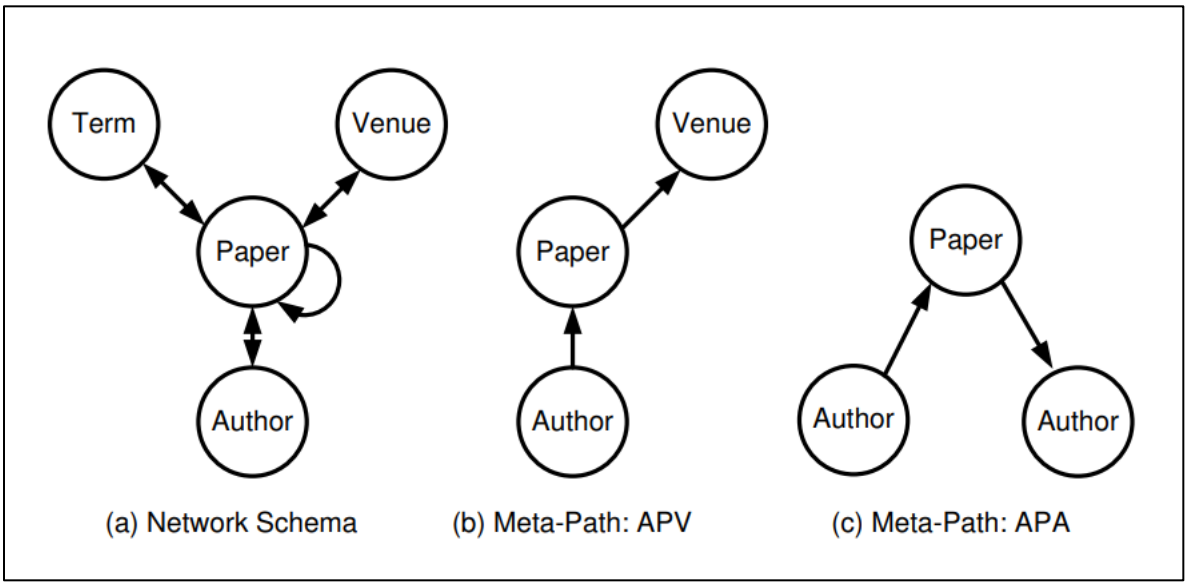


Figure 4. Bibliographic network schema and meta-paths [14]

For the example network schema Figure 4(a), two examples of meta-paths are given in 4(b) and 4(c). Through meta-paths, one can specify how different types are connected in a network. This algorithm can be considered as an extension to Node2Vec providing more flexibility to deal with heterogeneous graphs. It leverages the meta-paths to capture the semantic relationship between

different types of nodes and edges. Since metapath2vec allows to define meta-paths, domain-specific knowledge is also present that guides the random walks to explore relevant paths in the graph, thus capturing the desired structural patterns. Metapath2vec captures higher-order relationships, provides interpretable embeddings, and allows for more targeted exploration on heterogeneous graphs. The skip-gram model is applied to learn node embeddings based on the meta-path guided random walks.

3.6 Regular patterns on Knowledge Graphs

The graph embedding algorithms discussed so far primarily rely on random walk-based approach. However, there is another important characteristic that provides valuable insights into the underlying semantics and structure of the Knowledge Graph i.e., regular patterns. These patterns are the recurring structures that represent consistent relationships between entities (nodes). A regular pattern approach to generate node embeddings is provided by Keshavarzi et al. (2021), where a generalization of meta-paths is taken into consideration. A regular pattern as defined by Keshavarzi et al. (2021), is a schema subgraph that includes only the meta-nodes that are suitable for a specific problem.

3.7 RegPattern2Vec

RegPattern2Vec. RegPattern2Vec (Keshavarzi et al., 2021) is an algorithmic framework that focuses on capturing regular patterns in Knowledge Graphs. This algorithm incorporates the regular patterns into the embedding process. It is a method that takes advantage of the known schema to perform faster and more accurately by selecting only those nodes of the graph while traversal based on their types (guided by the regular pattern) in generation of random walks. The

RegPattern2Vec algorithm relies on random walks to produce embeddings and these random walks are constrained to those matching a defined regular pattern. In this algorithm, a regular pattern is converted to an equivalent Deterministic Finite Automata (DFA) [15], the finite automata steer the random walks and produces the random walks satisfying the regular pattern. The algorithm is highly scalable and performs effectively even on large Knowledge Graphs. The embeddings produced as a result of this algorithm capture the characteristics of each node and can be used to perform several downstream Machine Learning tasks such as node classification and link prediction.

Definition 1.4 (Deterministic Finite Automata). The formal definition of Deterministic Finite Automata (DFA) is it is a collection of 5-tuples ($Q, \sum, q_0, F, \delta$), where Q represents finite set of states, $\sum$ represents finite set of the input symbols, q_0 represents the initial state, F represents the final state and δ represents the transition function (δ: Q x $\sum \rightarrow Q$).

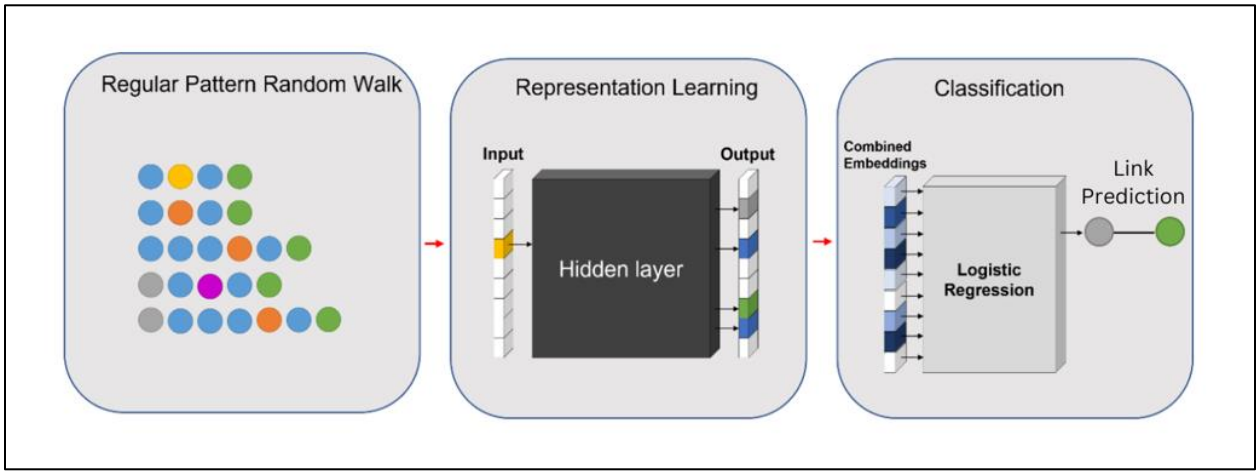


Figure 5. RegPattern2Vec generated random walks are fed as input to generate node embeddings, further used for link prediction (Keshavarzi et al., 2021)

To design meta-paths in metapath2vec, a domain expert is needed, and different meta-paths capture different connections between the nodes. And each meta-path requires a separate run as a separate experiment. RegPattern2Vec also overcomes this limitation as the regular pattern basically covers a large set of meta-paths like connections and can be run as a single experiment to capture the semantic connections. RegPattern2Vec makes use of the DFA to steer and generate random walks satisfying the regular expression. The usage of Finite Automata in RegPattern2Vec is explained in further detail in Chapter 4 where we propose the Neo4j implementation of RegPattern2Vec algorithm.

CHAPTER 4

NEO4J IMPLEMENTATION OF REGPATTERN2VEC

The proposed work in this thesis is the Neo4j implementation of the RegPattern2Vec algorithm in generation of random walks that can be used further to produce node embeddings. Broadly, the proposed work is implemented in Java and this section of the thesis comprises of the following.

1. Software architecture and libraries.
2. Proposed method.
3. Transforming regular expression to a transition table.
4. Input parameters.
5. Methodology.
6. Representation Learning on random walks.

4.1 System architecture and libraries

Performing the computations and generating the random walks that follow the regular pattern in Java is the core idea for this thesis. The implementation is carried out in IntelliJ IDEA Integrated Development environment [16], the project is built as a Gradle project and main libraries such as java.io, java.util, org.neo4j.graphdb, org.neo4j.procedure have been used as part of the implementation. Neo4j Desktop application is used for storing Knowledge Graphs and performing computations. The generated random walks are used as input to a model, similar to the one used in metapath2vec, for generating node embeddings.

The software versions used as part of the plugin implementation are documented below.

Software	**Version**
Neo4j Desktop	1.5.7
Neo4j library	5.7.0
APOC library	5.7.0
Neo4j Graph Data Science Library	2.3.5
Java	17
IntelliJ IDEA	2022.3.2 (Ultimate Edition)

4.2 Proposed method

The popularity of using Neo4j as a graph database has increasingly grown in the past few years. Factors such as agility, scalability, own graph query language, integration capability have contributed to the rise of this graph database. This thesis proposes an implementation of the RegPattern2Vec algorithm (Keshavarzi et al., 2021) as a custom plugin in the Neo4j graph database. The plugin generates random walks, which can then be used to generate vector embeddings. The plugin accepts a regular expression transformed Finite Automata as one of the inputs, and generates random walks based on this Finite Automata as output.

The generated random walks through this procedure effectively capture the semantic relationship between the graph nodes with minimum prior knowledge and human involvement. The plugin is then successfully tested on a citation network dataset and the results are documented.

4.3 Transforming regular expression to a transition table.

The transition table provides a deterministic representation of the regular expression. It represents the finite automata for the corresponding regular expression. Since each state and input symbol combination are well defined within the two-dimensional table format, the computations allow quick lookups and transitions between states, thereby improving the overall performance of the pattern matching. Conversion of a regular expression to a DFA involves usage of algorithms such as Thompson’s Construction. In this thesis, this conversion is hard-coded and the specific DFA

has been used all along in the algorithm. This is a scope of improvement in the custom plugin generated in this thesis. Since Java is used as a programming language of choice for this implementation, a DFA in the form of string has been used for the corresponding regular expression input. For example, for regular expression, **(a)(b)*(c)**, the equivalent string DFA that would be considered is **((q_0, a→q_1), (q_1, b→q_1|c→q_2))**. Here **q_0**, **q_1**, **q_2** represent the states of DFA, **a**, **b**, **c** represents the input symbols (node types). The equivalent transition table considered for the above DFA would be as depicted in Figure 6. In general, the regular expression can include any node types and any operators of regular expression and the plugin can operate accordingly.

		Input States		
		a	b	c
States	q_0	q_1	-	-
	q_1	-	q_1	q_2
	q_2	-	-	-

Table 1. Transition table for DFA.

The – indicates that there is no possible transition to the next state for the considered input symbol from the current state. The transition table clearly explains that the start node q_0 upon receiving a as input symbol, the next state it would transition to is q_1. Similarly, in q_1 state, if b is received as input symbol, it stays in the same state q_1, and when it receives c as input symbol would transition to next state q_2. Here q_2 is the final state and no further transitions are allowed. The transition table clearly depicts the formulated regular expression (a)(b)*(c) and the DFA **((q_0, a→q_1), (q_1, b→q_1|c→q_2))**.

The transition table in this proposed method acts as an important element for steering the random walks and generating them. To make it programmatically easier and computationally fast, we assume the states as integers and the input symbols as integers as well. So, Figure 6 can be expressed in the form of a numerical transition table, where the input symbols a, b, c is denoted as 0, 1, 2 and the states q0, q1, q2 are represented as 0, 1, 2 respectively. No possible transition is depicted using -1. Figure 7 shows the transition table for the same.

		Input States		
		0	1	2
	0	1	-1	-1
States	1	-1	1	2
	2	-1	-1	-1

Table 2. Considered transition table for DFA.

4.4 Input parameters

The first step in the implementation of the algorithm is the consideration of the input parameters for the algorithm. We have considered three input parameters: a regular expression, walk length, number of walks.

regular expression. A regular expression of the format, for example, **(a)(b)*(c)** is considered, where **a, b, c** denotes the node types of the respective Knowledge Graph. The expression encapsulates the node types with simple brackets. The epsilon star (*****) implies zero or more occurrences of the node type enclosed with it. The regular expression basically is a set of meta-paths generalized into one format.

walk length. The walk length refers to the number of steps or nodes visited in each random walk. It determines how far a random walk will explore the graph from the starting node. The length of

the random walk determines how much information is captured about the local neighborhood of each node.

number of walks per node. It refers to how many random walks are generated for each starting node. Performing multiple walks per node captures different neighborhood structures by traversing different possible paths.

An example input to the algorithm:

Regular expression: **(a)(b)*(c)**, walk length: **10**, number of walks per node: **5**.

Here, the input implies that we intend to get the random walks each of length **10** and number of walks per each starting node is **5** that satisfy the regular expression **(a)(b)*(c)**.

4.5 Methodology

The algorithm implementation assumes that the provided DFA is the basis with which the random walks are generated. The goal is to only traverse those nodes that belong to the specific node type and generate random walks of specified walk length that satisfy the regular expression. This methodology implements an iterative solution to produce random walks.

Step 1. The approach starts by finding all the nodes (starting nodes) that satisfy the regular expression. For example, for regular expression (a)(b)*(c), the method finds all nodes of the node type **a**. Here onwards, the iterative solution starts where we need to produce the random walks starting with these nodes of the specified walk length and the number of walks per each starting node.

Step 2. To traverse to the next node, we initially fetch the node types that are attached to the current node by an edge. Neo4j.graphdb library allows the traversal of the graph nodes and perform logical operations on the Knowledge Graph. Then, the valid nodes that belong to the specific node types are collected. One node at random is selected as the next node. On large

Knowledge Graphs, some nodes have a high number of incoming and outgoing edges. Hence, there would be domination of those nodes in the random walks as they have higher probability of being reached when the next node is selected. The walks are biased by adapting a part of the formula below, similar to as defined in the metapath2vec paper.

$$p(v^{i+1}|v_t^i, \mathcal{P}) = \begin{cases} \frac{1}{|N_{t+1}(v_t^i)|} & (v^{i+1}, v_t^i) \in E, \phi(v^{i+1}) = t+1 \\ 0 & (v^{i+1}, v_t^i) \in E, \phi(v^{i+1}) \neq t+1 \\ 0 & (v^{i+1}, v_t^i) \notin E \end{cases}$$

In the formula, $|N_v|$ implies the degree of the node v, v^i_t indicates the current node and v^{i+1} denotes the next candidate node, $N_{t+1}(v^i_t)$ denotes the V_{t+1} type of neighborhood of node v^i_t, DFA is ($\mathcal{P}$). When there are no edges, or when there is an edge, but the node type of the candidate node is not matching the node type of the next input symbol (node type) of the regular pattern (DFA), the probability of selection of v^{i+1} is 0. When there is an edge and satisfying node type, the degree of the node determines the probability of the node selection. In this implementation, all these valid nodes are stored in a Node Iterator and then a node is picked based on the implemented Biased Pick method that follows the formula above. This consideration tends to pick those nodes which have degree. An improvement that can be incorporated for the current Neo4j implementation of the RegPattern2Vec algorithm is to follow the two-stage selection process where we initially choose the node type among the possible node types and then choosing the corresponding node. This has been implemented in the RegPattern2Vec python implementation. This step can avoid choosing the node of the same node type repeatedly.

Step 3. An important factor to take into consideration in this implementation is to use the reverse regular expression (in turn its related DFA and the transition table). The reason to do this

is for scenarios where the traversal has reached the final state, however the walk length has not reached yet. The reverse regular expression helps to continue the walk. This is repeated until the walk length is reached. For example, for (a)(b)*(c), the equivalent regular expression would be (c)(b)*(a). This also ensures that we are still abiding by the regular pattern logic, and we will be traversing only those nodes of the node types satisfying the regular pattern and connected accordingly. In the implementation, a boolean flag **forw** is considered. When forw is true, it implies that we are following the original regular expression's transition table. When the transition table reaches its' final state, the boolean flag **forw** is set to false and then the traversal continues on the reverse regular expression's transition table **(Figure 8)**, and vice versa. This is repeated until the walk length is reached in producing the random walk.

		Input States		
		c	b	a
States	q_0	q_1	-	-
	q_1	-	q_1	q_2
	q_2	-	-	-

	Input States			
	c	b	a	
States	0	1	-1	-1
	1	-1	1	2
	2	-1	-1	-1

Table 3. Reverse transition table for the regular expression of the format (a)(b)*(c)

Step 4. The iterative code repeats until the random walks following the regular expression of specified walk length and specified number of walks per each starting node is reached. The result is random walks, which is the output of this implementation. These random walks are stored into a text file.

Few other important considerations in the implementation of this algorithm include the use of Map data structure, where possible to reduce the computational complexity in traversing the nodes, implementing the iterative solution instead of recursive solution, since the recursive

solution involves the computational overhead on graph structures and would result in a delayed output on large Knowledge Graphs, thus affecting scalability.

4.6 Representation Learning on random walks

Representation Learning on random walks is a technique to automatically discover meaningful representations from random walks. The goal is to capture the structural and semantic information of the random walks of Knowledge Graph and to transform data into a meaningful representation that captures this underlying data. This custom plugin implementation of RegPattern2Vec produces random walks (sequence of nodes). These are treated as input to a model, similar to the one used in metapath2vec [17] for generating node embeddings. The model used is an improved version of the original skip-gram model as the types of nodes are taken into consideration. The node embeddings produced as a part of this model act as an input to machine learning algorithms and can be used for experiments such as node classification and link prediction.

CHAPTER 5

EXPERIMENTS

5.1 Datasets

In our experiments, we have used a subset of the citation network dataset [18], a dataset extracted from DBLP, ACM, and MAG. The subset dataset data files can be found on my google drive [19]. The subset dataset is focused on articles from four publications namely, Lecture Notes in Computer Science, Communications of the ACM, International Conference on Software Engineering, and Advances in Computing and Communications. The Knowledge Graph built with this subset dataset comprises of three node types Author, Articles, and Venue. The subset dataset contains 80,299 authors, 184,313 articles, 4 venues, 140,575 author relationships and 289,908 citation relationships.

We made use of another subset dataset of Olympic Results Dataset [20], a historical dataset on the modern Olympic games and performed experiment on the generated node embeddings. This subset dataset is focused on three node types namely Athlete, Participation and Medal among many other node types that exist in the Knowledge Graph such as Team, Game, Event, Sport. Athlete node type contains the properties of the athlete such as name, gender, height and weight. Medal node type contains three possible values: Gold, Silver, and Bronze. Participation node type represents the participation of an athlete in an event at a game for a team with an optional medal. This information is presented as relations to the appropriate nodes and properties and the Knowledge Graph is built.

5.2 Building the Knowledge Graph.

We begin our experiment by creating a Knowledge Graph for the citation network subset dataset. Cypher query language in Neo4j allows us to build this Knowledge Graph. The following cypher query loads the JSON data to populate the Knowledge Graph with Article, Author, and Venue nodes, and also establishes the respective relationships as described in the data and the query. APOC library [21] provides apoc.load.json procedure to load the JSON data. Uniqueness constraint on the three node types is maintained in order to prevent duplicate nodes.

```
CREATE CONSTRAINT FOR(a:Article) REQUIRE a.index IS UNIQUE;
CREATE CONSTRAINT FOR (a:Author) REQUIRE a.name IS UNIQUE;
CREATE CONSTRAINT FOR (v:Venue) REQUIRE v.name IS UNIQUE;
CALL apoc.periodic.iterate(
  'UNWIND ["dblp-ref-0.json", "dblp-ref-1.json", "dblp-ref-2.json", "dblp-ref-3.json"]
   AS file
   CALL apoc.load.json
   ("https://drive.google.com/drive/folders/11EY36NpkOAm88QrDirsWWdMW1kvO9xJ2?usp=sharing"
   + file)
   YIELD value WITH value
   RETURN value',
  'MERGE (a:Article {index:value.id})
   SET a += apoc.map.clean(value,["id","authors","references", "venue"],[0])
   WITH a, value.authors as authors, value.references AS citations, value.venue AS venue
   MERGE (v:Venue {name: venue})
   MERGE (a)-[:VENUE]->(v)
   FOREACH(author in authors |
     MERGE (b:Author{name:author})
     MERGE (a)-[:AUTHOR]->(b))
   FOREACH(citation in citations |
     MERGE (cited:Article {index:citation})
     MERGE (a)-[:CITED]->(cited))',
   {batchSize: 1000, iterateList: true});
```

Figure 6. Cypher query to build a Knowledge Graph on a subset citation network dataset.

The cypher query in **Figure 6** builds the Knowledge Graph with Author, Article, Venue node types. It establishes AUTHOR, CITED, VENUE relationships as well between the nodes. **Figure 7** illustrates the part of the Knowledge Graph that is created with the above cypher query. The nodes in green are the Venue nodes, nodes in red are article nodes and the nodes in brown are author nodes.

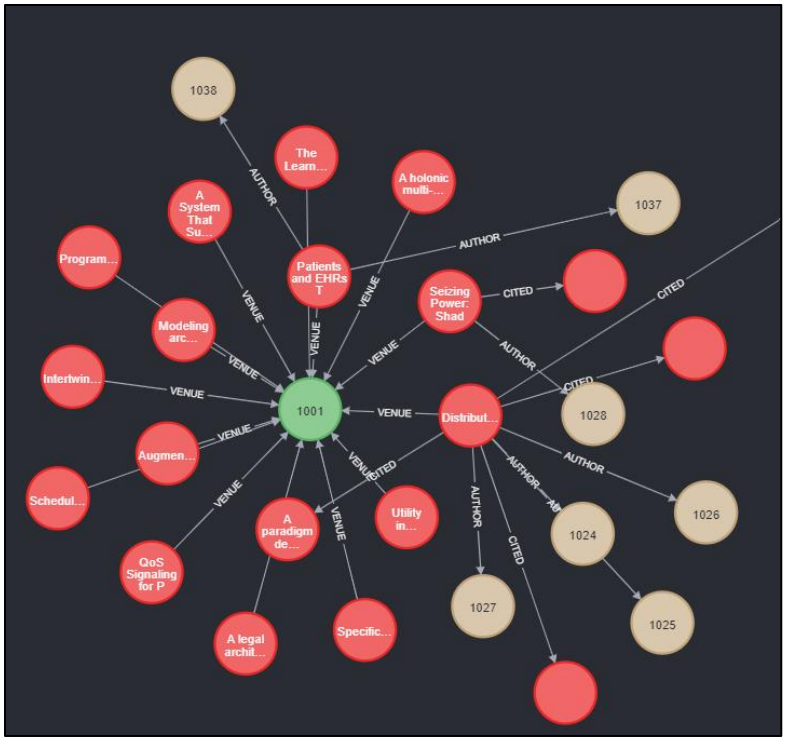
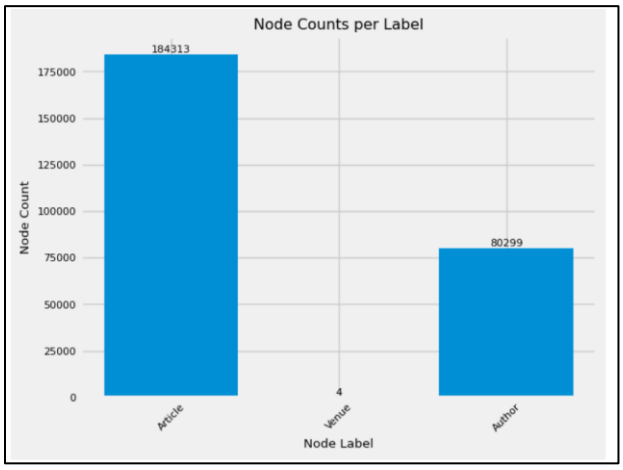


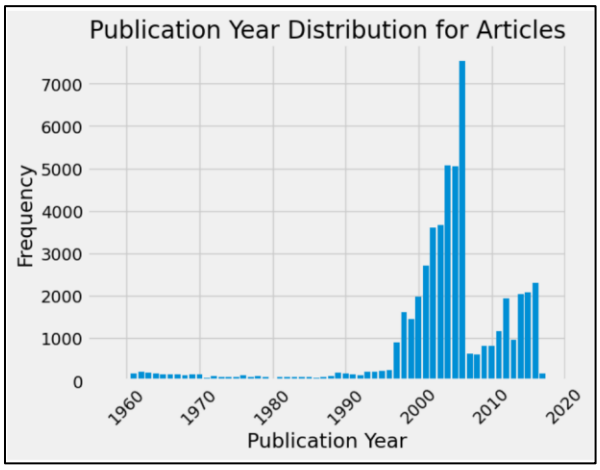
Figure 7. Part of the Knowledge Graph that is created.

5.3 Understanding the Knowledge Graph.

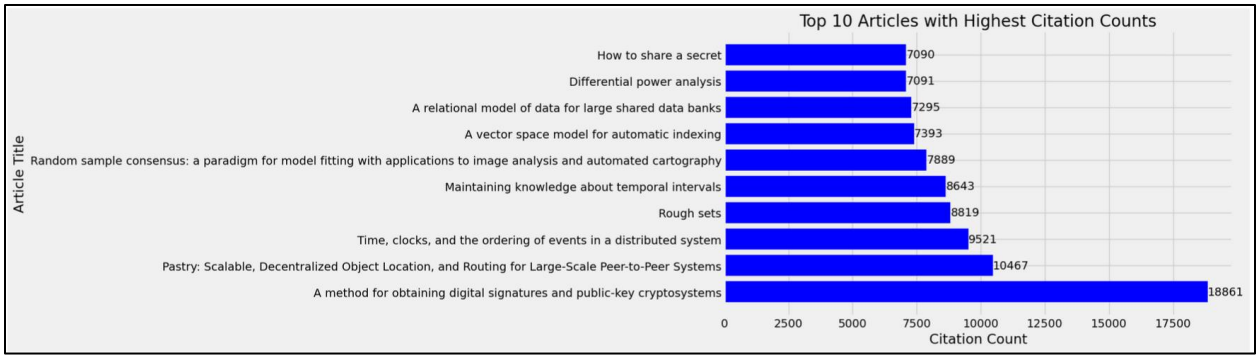
To gain insights on the graph schema and understanding the characteristics of the Knowledge Graph(KG), we perform data analysis with the help of cypher queries and python on the KG.



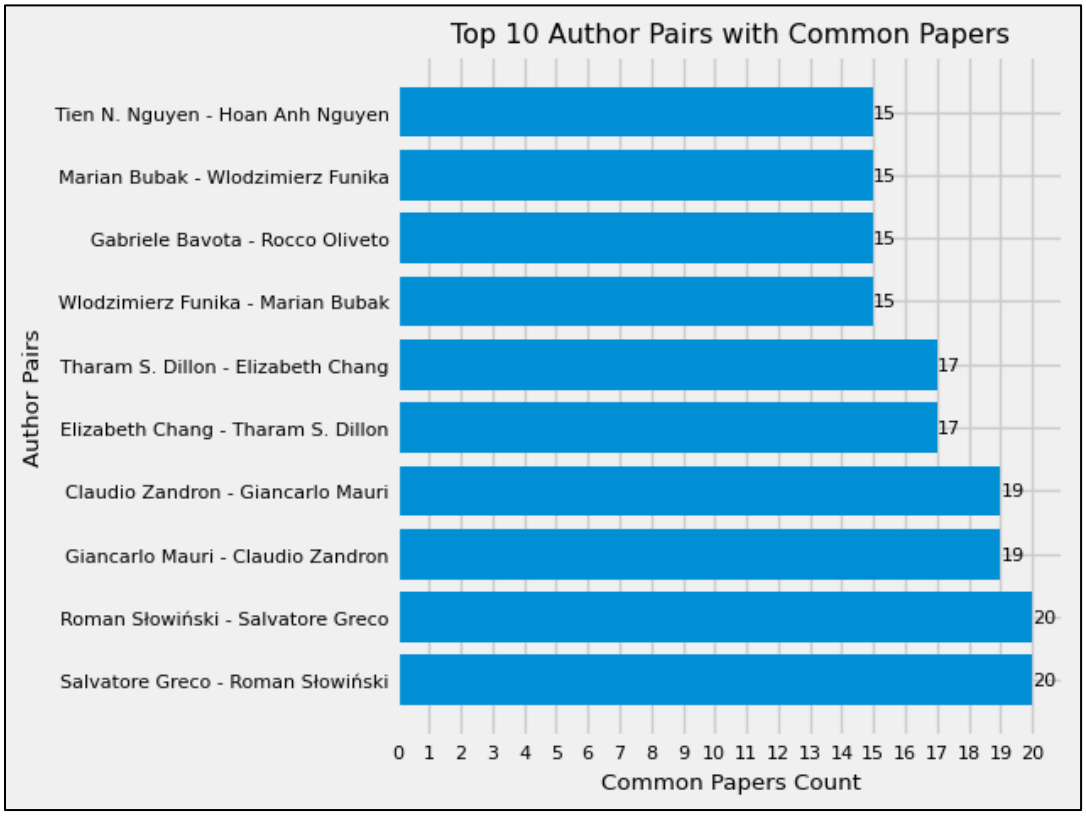
a. Node counts per node type



b. Publication year distribution for Articles.



c. Top 10 Articles with Highest Citation Counts



d. Top 10 author pairs with common articles

Figure 8 (a.) depicts the number of nodes for each node type, (b.) depicts the publication year for the articles, (c.) depicts the top 10 articles that have the highest number of citations and (d.) indicates the top 10 author pairs that have the most articles in common from the Knowledge Graph built in Neo4j.

5.4 Regular expression input for the Neo4j implementation of RegPattern2Vec

From the Knowledge Graph, as a part of our experiment, we considered the regular expression of the form **(Author)(Article)*(Venue)**. This regular expression is a pattern that considers the meta-paths that considers exact node type Author, followed my zero or more occurrences of the node type Paper, followed by Venue node type. Few possible meta-paths satisfying the regular expression are:

1. (Author)(Article)(Venue),
2. (Author)(Article)(Article)(Venue),
3. (Author)(Article)(Article)(Article)(Venue), and so on.

The regular expression basically considers all the possible meta-paths that satisfy the regular expression, which is one of the advantages of employing the RegPattern2Vec algorithm over the metapath2vec algorithm.

5.5 Generating random walks using the custom RegPattern2Vec plugin.

In order to generate random walks, the following input parameters have been considered for the initial experiment, regular expression = **(Author)(Article)*(Venue)**, walk length = **10**, number of walks per node = **10**. Taking these parameters as input, the regular expression is transformed (within the code manually) to the Deterministic Finite Automata (DFA) string of the form **((q_0, Author$\rightarrow q_1$), (q_1, Article$\rightarrow q_1$|Venue$\rightarrow q_2$))**. These inputs when fed into the Neo4j implemented RegPattern2Vec algorithm produce random walks each of walk length 10 and the number of walks per each starting node are 10. All these random walks satisfy the regular expression pattern provided. The random walks are stored in a syntax structure in such a way that it is compatible to serve as input to the improved version of the skip-gram model in metapath2vec algorithm to

generate the node embeddings. A sample random walk generated using the Neo4j implemented RegPattern2Vec plugin is documented below.

a230762 i227109 v1031 i262051 a91421 i88866 i91425 i116550 i119043 i116550

In this random walk, '**a**' represents that the node is of author node type, '**i**' indicates that the node is of article node type, '**v**' indicates that the node is of venue node type. The numbers concatenated with these node types are the node Ids of the corresponding nodes. The generated random walks follow the input provided strictly and adhere to generating the fixed length random walks as specified. If the final state is reached during traversal for the DFA **($(q_0$, Author$\rightarrow q_1)$, $(q_1$, Article$\rightarrow q_1$\|Venue$\rightarrow q_2))$**, the algorithm applies the DFA of the reverse regular expression **($(q_0$, Venue$\rightarrow q_1)$, $(q_1$, Article$\rightarrow q_1$\|Author$\rightarrow q_2))$**and this repeats until the walk length is reached. The generated random walks are stored as a text file locally. Traversing a regular expression abided Knowledge Graph helps in avoiding unwanted traversals and focuses only on those nodes that are of interest for the particular problem.

5.6 Generating node embeddings based on the random walks.

Generation of random walks concludes the Neo4j implementation of the RegPattern2Vec algorithm. The output of the algorithm, the random walks are then fed as input to the model [22] similar to the one used in metapath2vec to generate node embeddings. The input format is already in alignment and taken care of in the random walk generation step. The random walks are stored in a text format and then passed into the model. The output of this model is the node embeddings. These node embeddings capture the structural and semantic information of the random walks, thereby capturing the underlying information of the Knowledge Graph under test.

5.7 Link Prediction experiment on the generated node embeddings.

5.7.1 Predicting if there exists a citation relationship between two article nodes.

Link Prediction is an interesting topic in the field of Knowledge Graphs. It involves predicting the missing or the future connections between entities in a Knowledge Graph. An experiment on the generated node embeddings is performed to predict if there exists a citation relationship between two articles. The node embeddings represent the low-dimensional vectors and store the underlying information of the node structure and the graph data. Performing this experiment on the generated node embeddings from the custom Neo4j implemented RegPattern2Vec algorithm provides a good understanding on how well the generating node embeddings function with respect to the link prediction experiment.

Step 1. Load the node embeddings.

We make use of several python libraries as a part of this experiment such as sklearn [23], numpy [24], matplotlib [25], py2neo[26], pandas[27], pyspark[28], matplotlib [29]. The generated node embeddings are loaded into the Jupyter [30] notebook. After loading, the node Ids and their corresponding embeddings are stored as lists in **node_ids** and **embeddings** variables respectively.

Step 2. Retrieve citation relationships between articles.

The generated node embeddings are stored in a text file. These node embeddings predominantly contain the article node embeddings as the regular expression we have used for generating random walks is **(Author)(Article)*(Venue)**. As the experiment we run on requires the use of these node embeddings, we make use of two cypher queries to create two data frames that depict the citation relationship between article pairs. If there exists a citation relationship between two article nodes, it is labeled as **1**. The second cypher query helps identify pairs of articles that have common citation relationships but are not directly connected. These are labeled as **0**.

```
MATCH (a1:Article)-[:CITED]->(a2:Article)
RETURN id(a1) AS node1, id(a2) AS node2, 1 AS label
```

(a)

```
MATCH (a1:Article)-[:CITED]-(a2:Article)
WHERE NOT ((a1)-[:CITED]->(a2))
RETURN id(a1) AS node1, id(a2) AS node2, 0 AS label
```

(b)

	node1	node2	label
0	0	1009	1
1	0	1008	1
2	0	1007	1
3	0	1006	1
4	0	1005	1
...	...	...	...
289903	262739	264601	1
289904	262739	249680	1
289905	262739	54197	1
289906	262739	264604	1
289907	262739	264275	1

289908 rows × 3 columns

(c)

	node1	node2	label
0	8	167055	0
1	18	133977	0
2	18	70335	0
3	22	116496	0
4	27	63702	0
...	...	...	...
289407	264611	262739	0
289408	264612	262739	0
289409	264613	262739	0
289410	264614	262739	0
289411	264615	262739	0

289412 rows × 3 columns

(d)

Figure 9 (a.) article pairs with direct cited relationship (b.) article pairs with indirect citation relationship (c.) data frame of article pairs with label **1** (d.) data frame of article pairs with label **0**

These data frames contain the citation relationship information between article pairs. One data frame contains article pairs that are directly connected. Another data frame contains the article pairs that are one hops away from each other (not directly connected) based on the citation relationship. The cypher query 9 (b) retrieves article pairs that have an indirect citation relationship, meaning that they are not directly connected by a CITED relationship, but have a common article between them.

Step 4. Extract node embeddings from the node embeddings file

With the data frames in hand, our goal is to extract node embeddings of the article pairs that are present in the generate node embeddings text file. To perform this, we iterate over the data frames, compare the nodes and extract the **node_id** pairs from the embeddings file. We then extract the relevant node embeddings of the node id pairs from the **embeddings** list.

1. We extract node embeddings for the article pairs from Figure 13 (c) data frame. The extracted node embedding pair is concatenated to form one embedding, which is then added to a list of connected embeddings. The concatenation operation results in a new vector that contains the combined information from both nodes, capturing a more comprehensive representation of the relationship between the nodes. As a result of this step, for our experiment, we got **264,382** embeddings that represent the connected node pairs (label = 1).

2. We repeat the same procedure as the above step for the article pairs from Figure 9 (d) data frame. The extracted node embedding pair is concatenated to form one embedding, which is then added to another list of embeddings that represent the not connected embeddings. As a result of this step, for our experiment, we got **263,886** embeddings that represent the not connected node pairs (label = 0).

Now, we have the required embeddings to perform model training and run experiments on our embeddings. In total, for this experiment we have **528,268** embeddings on which we can perform model training. We create test, train sets on these embeddings and evaluate the results.

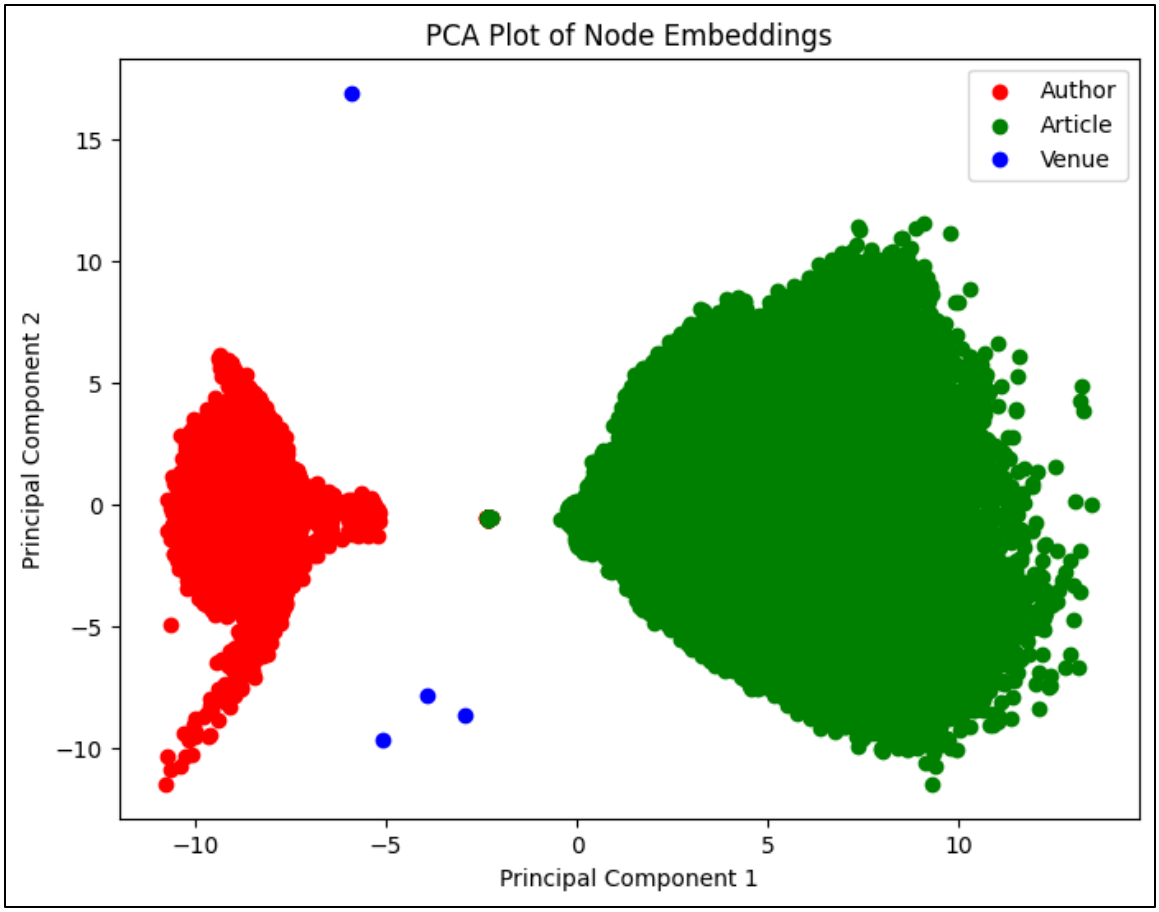


Figure 10. Principal Component Analysis plot of the node embeddings

Figure 10 shows the principal component analysis plot of the node embeddings. Principal Component Analysis [31] is a dimensionality reduction technique used to visualize data. PCA allows to plot the embeddings in a lower dimensional space and visually inspect clusters.

Step 5. Training a model

Logistic regression [32] on node embeddings refers to applying the logistic regression algorithm on node embeddings as input features for binary classification tasks. This model leverages these learned representations to predict the binary label or class of nodes. The node embeddings are the input features for the model, which then learn the relationship between the embeddings and the target variable. Once trained, the model can be used to make predictions on the data.

We have **264,382** node embeddings that represent connected pairs, and **263,886** node embeddings that represent node embeddings that are not connected to each other directly with a relationship. This is more of a balanced set. We consider a relatively balanced set of **528,268** node embeddings. We shuffle this balanced set of node embeddings and then train the model on these embeddings. We initially split this balanced set of node embeddings data into training and testing sets in a **80:20** split, where 80 percent is the training set (**422,614**) and 20 percent is the testing set (**105,654**). The accuracy, recall, f1 score, roc_auc score for the trained model on this data is recorded as follows.

Metric	**Score**
Accuracy	0.9181573816419634
Recall	0.9181344203242013
f1 score	0.9180386915763832
roc_auc score	0.918157346055805

Table 4. Performance metrics on the trained model for predicting if citation relationship exists between two article nodes.

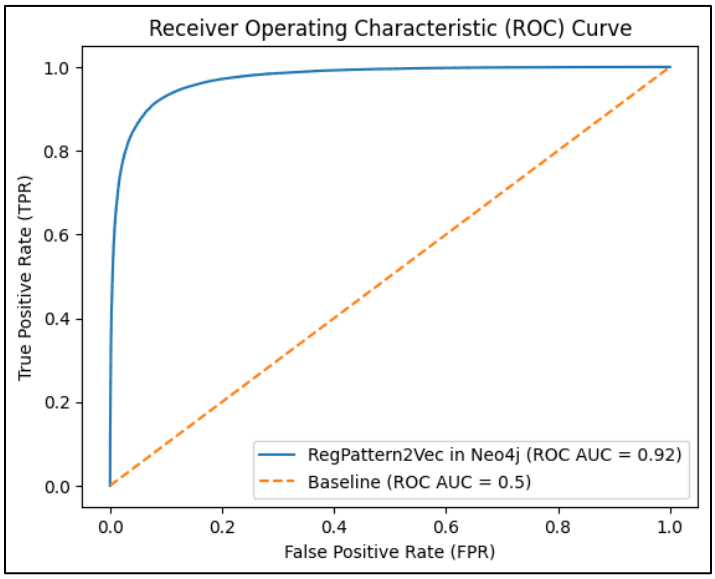


Figure 11. ROC curve of the link prediction experiment

We then ran a prediction experiment that provides two different nodes of the article node type and predicted the citation relationship between the two.

```
node_ids = ['1000', '1002']
node1_id, node2_id = node_ids[0], node_ids[1]

node1_embedding_run = new_embeddings[node1_id]
node2_embedding_run = new_embeddings[node2_id]

node_embeddings = np.concatenate([node1_embedding_run, node2_embedding_run])
node_embeddings = node_embeddings.reshape(1,-1)
prediction = model.predict(node_embeddings)

# Interpret the prediction
if prediction == 1:
    print("A citation relationship is predicted between the two nodes.")
else:
    print("No citation relationship is predicted between the two nodes.")
```

```
A citation relationship is predicted between the two nodes.
```

Figure 12. Link Prediction to determine if citation relationship exists.

With the walk length of 10 and number of walks per each starting node as 10, the generated node embeddings from our custom RegPattern2Vec algorithm provide scores as in Table 4 with Logistic Regression. Since walk length and number of walks per each starting node are two important parameters in generation of the random walks and then the node embeddings, the citation prediction experiment is run on increasing walk length from 10 to 20 while maintaining the number of walks per each starting node as 10, and another experiment increasing the number of walks per each starting node from 10 to 20 while maintaining the walk length as 10 to see how the model behaves with the changes. The results are documented as follows:

For Walk Length: 20, Number of walks: 10,

Metric	**Score**
Accuracy	0.9266645783300268
Precision	0.9276725186859031
Recall	0.9259664098827122
f1 score	0.926818679124047
roc_auc score	0.9266666988667854

Table 5. Performance metrics on the trained model for predicting if citation relationship exists between two article nodes. for walk length of 20, number of walks per each starting node of 10

For Walk Length: 10, Number of walks: 20,

Metric	Score
Accuracy	0.9221791947376797
Precision	0.921577963531196
Recall	0.9230755643480412
f1 score	0.9223261560183551
roc_auc score	0.9221782268387518

Table 6. Performance metrics on the trained model for predicting if citation relationship exists between two article nodes. for walk length of 10, number of walks per each starting node of 20

The result shows a minor increase in the metrics in both the scenarios compared to the initial walk length and number of walks scores. An increase of walk length from 10 to 20 has shown more improved result than that of increasing number of random walks per each starting node from 10 to 20.

5.7.2 Predicting future citation probability between two articles.

We repeat the same experiment as **5.7.1** for predicting the future citation probability based on the node embeddings generated. From Figure 8 (b), we can see that in the year 2006, we had the most number of publications. This can act as a good separating point for preparing the positive and negative labels. We use the cypher queries as depicted in Figure 13.

```
MATCH (a1:Article)-[:CITED]->(a2:Article)
WHERE a1.year <= 2006
RETURN id(a1) AS node1, id(a2) AS node2, 1 AS label
```

(a)

```
MATCH (a1:Article)-[:CITED]->(a2:Article)
WHERE a1.year > 2006
RETURN id(a1) AS node1, id(a2) AS node2, 0 AS label
```

(b)

	node1	node2	label
0	0	1009	1
1	0	1008	1
2	0	1007	1
3	0	1006	1
4	0	1005	1
...	...	...	...
182064	262687	52768	1
182065	262687	103620	1
182066	262687	193180	1
182067	262687	52769	1
182068	262687	27951	1

182069 rows × 3 columns

(c)

	node1	node2	label
0	1	1025	0
1	1	1024	0
2	1	1023	0
3	1	1022	0
4	1	1021	0
...	...	...	...
107834	262739	264601	0
107835	262739	249680	0
107836	262739	54197	0
107837	262739	264604	0
107838	262739	264275	0

107839 rows × 3 columns

(d)

Figure 13 (a.) article pairs with citations before 2006(inclusive) (b.) article pairs with citations after 2006 (c.) data frame of article pairs with label **1** (d.) data frame of article pairs with label **0**

These data frames contain the citation relationship information between article pairs. One data frame contains article pairs that have cited relationship before 2006 (including 2006). Another data frame contains the article pairs that have cited relationship after 2006. We extract node embeddings similar to the above experiment and have 171,108 node embeddings that represent the node embedding pair of articles before and including year 2006. We have 93,274 node embeddings that represent the node embedding pair of articles after the year 2006. We consider the combined set of node embeddings and then train the model on these embeddings. We initially split this set of node embeddings data into training and testing sets in an **80:20** split, where 80 percent is the training set (211,505) and 20 percent is the testing set (52,877).

The accuracy, recall, f1 score, roc_auc score for the trained model on this data is recorded as follows.

Metric	Score
Accuracy	0.895568961930518
Precision	0.9008515138023152
Recall	0.9455613348138624
f1 score	0.9213838268792711
roc_auc score	0.8747104449464649

Table 7. Performance metrics on the trained model for predicting a future citation relationship between two article nodes.

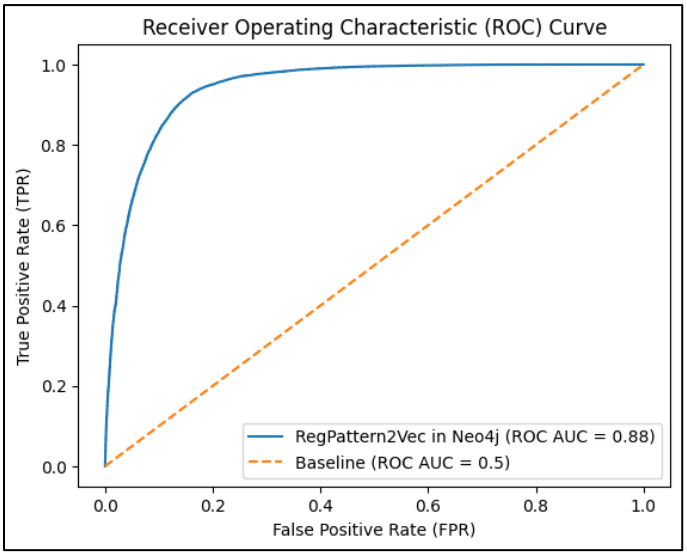


Figure 14. ROC curve of the citation prediction experiment

We ran a prediction experiment to predict the likelihood of a citation between two article pairs, whether they are cited before or after 2006 and predicted the relationship between the two.

```
node_ids = ['100', '1022']
node1_id, node2_id = node_ids[0], node_ids[1]

node1_embedding_run = new_embeddings[node1_id]
node2_embedding_run = new_embeddings[node2_id]
```

```
node_embeddings = np.concatenate([node1_embedding_run, node2_embedding_run])
node_embeddings = node_embeddings.reshape(1,-1)
prediction = model.predict(node_embeddings)

# Interpret the prediction
if prediction == 1:
    print("There is a high likelihood of a future citation between the two nodes.")
else:
    print("There is a low likelihood of a future citation between the two nodes.")
```

```
There is a low likelihood of a future citation between the two nodes.
```

Figure 15. Link prediction experiment to determine future citation.

With the walk length of 10 and number of walks per each starting node as 10, the generated node embeddings from our custom RegPattern2Vec algorithm provide scores as in Table 7. Since walk length and number of walks per each starting node are two important parameters in generation of the random walks and then the node embeddings, the future citation prediction experiment is run on increasing walk lengths and number of walks per each starting node. The results are documented as follows:

For Walk Length: 20, Number of walks: 20,

Metric	**Score**
Accuracy	0.9143059490084986
Precision	0.9157367402655923
Recall	0.9509496284062758
f1 score	0.9213838268792711
roc_auc score	0.8747104449464649

Table 8. Performance metrics on trained model for predicting a future citation relationship between two article nodes for walk length of 20, number of walks per each starting node of 20

For Walk Length: 30, Number of walks: 30,

Metric	**Score**
Accuracy	0.9194121502742609
Precision	0.9216104519397413
Recall	0.952282557787145
f1 score	0.9366954825072492
roc_auc score	0.9083274670192356

Table 9. Performance metrics on the trained model for predicting a future citation relationship between two article nodes for walk length of 30, number of walks per starting node of 30

For Walk Length: 40, Number of walks: 40,

Metric	Score
Accuracy	0.9224759408092167
Precision	0.9230240731443068
Recall	0.9566717791411042
f1 score	0.9395467688790262
roc_auc score	0.9104970446706918

Table 10. Performance metrics on the trained model for predicting a future citation relationship between two article nodes for walk length of 40, number of walks per each starting node of 40

The results show an increase in the metric scores with an increase in the walk length and number of walks per starting node as depicted in Figure 16. These two parameters capture the local context and the neighboring structure and effectively understand the structural information of the nodes.

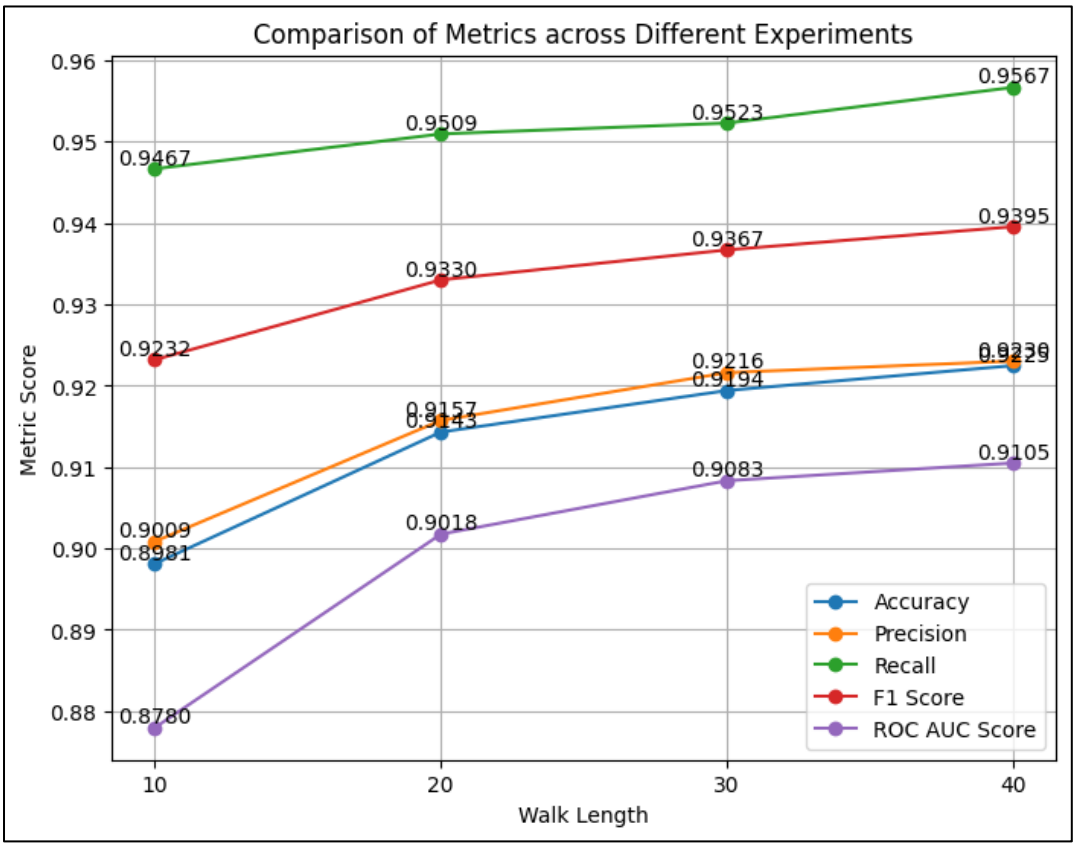


Figure 16. Comparison of metrics across different experiments with varied walk lengths and number of walks per each starting node

5.8 Node classification experiment on 120 years of Olympic Results Dataset

We begin the experiment by creating the Knowledge Graph of the Olympic Results Dataset. The Knowledge Graph consists of Athlete, Participation, Medal, Team, Game, Event, Sport node types and have HAS_ATHLETE, HAS_MEDAL, HAS_TEAM, HAS_GAME, HAS_EVENT relationships between Participation and Athlete, Medal, Team, Game, Event nodes respectively.

From the Knowledge Graph, as a part of experiment, we considered three node types from the graph, namely Athlete, Participation and Medal. The regular expression of the form **(Athlete)(Participation)*(Medal)** is taken as one of the inputs to the Neo4j implemented RegPattern2Vec algorithm. To generate random walks, the following input parameters have been considered for this experiment, regular expression = **(Athlete)(Participation)*(Medal)**
walk length = 10, number of walks per node = 10. Taking these parameters as input, the regular expression is transformed (within the code manually) to the Deterministic Finite Automata (DFA) string of the form **((q0, Athlete→q1), (q1, Participation→q1|Medal→q2))**. These inputs when fed into the Neo4j implemented RegPattern2Vec algorithm produce random walks each of walk length 10 and the number of walks per each starting node as 10. These random walks are then fed into the model similar to the one used in metapath2vec to generate node embeddings. These node embeddings capture the structural and semantic information of the random walks, thereby capturing the underlying information of the Knowledge Graph under test.

Similar to the link prediction experiment in 5.7.1, a node classification experiment is performed on the generated node embeddings of the 120 years Olympic Results subset dataset to determine if the athlete is a medal winner or not. The node embeddings are loaded into the Jupyter notebook and corresponding embeddings are stored as list. We make use of cypher query to create data frame that depict if the athlete is a medal winner or not. If the participation node and the

athlete node has a Gold, Silver, or Bronze medal, they are considered as positive labels. If the participation and athlete node that has no medal, they are considered as negative labels.

```
df1 = graph.run("""
MATCH (p:Participation)-[:HAS_MEDAL]->(m:Medal)
MATCH (p)-[:HAS_ATHLETE]->(a:Athlete)
RETURN DISTINCT id(p) AS participationId, id(a) AS athleteId,
  CASE m.type
    WHEN 'Gold' THEN 'Gold Medalist'
    WHEN 'Silver' THEN 'Silver Medalist'
    WHEN 'Bronze' THEN 'Bronze Medalist'
    ELSE 'No Medal'
  END AS label
  """).to_data_frame()
```

	participationId	athleteId	label
0	137946	162	No Medal
4	406760	135292	No Medal
9	407244	135507	No Medal
14	407017	135408	No Medal
16	406726	135283	No Medal
...	...	...	...
269726	137747	37	Silver Medalist
269727	137952	164	Silver Medalist
269728	138544	513	Silver Medalist
269729	138231	338	Silver Medalist
269730	137864	106	Silver Medalist

117863 rows × 3 columns

Figure 17. (a) Cypher query for participation and athlete pairs (b) data frame with the participation and athlete pairs and the corresponding medal won.

From the data frame, we can see that we have 117,863 participation and athlete pairs. Of them, 16,467 rows represent the medal winner (Gold, Silver, or Bronze) pair and the remaining 101,396 rows represent no medal pair. The corresponding node embeddings are then extracted from the node embeddings file. We extract the participation and the corresponding athlete pair's node embeddings and concatenate to form one embedding, which is then added to a list of connected embeddings. The concatenation operation results in a new vector that contains the combined information from both nodes, capturing a more comprehensive representation of the relationship between the nodes. We then perform Logistic Regression model training on these embeddings and evaluate the results.

With 16,467 embeddings representing medal winners and 101,396 embeddings representing no-medal winners, there is a large class imbalance between medal and no medal winners. Since the usecase is classifying medal winners and no medal winners, and that there will be class imbalance irrespective of the dataset size (since medal winners will always be less than the no medal winners), we applied Synthetic Minority Oversampling Technique (SMOTE)[33] on the generated embeddings. SMOTE addresses the class imbalance problem in Machine Learning datasets. The SMOTE algorithm works by creating synthetic samples of the minority class, the class with fewer samples, in this case, the medal winners. It does this by generating synthetic examples that are interpolated between existing minority class instances, thus handling the class imbalance problem. Later, we trained the model on these embeddings in a 80:20 train-test split. The accuracy, recall, f1 score, roc_auc score for the trained model on this data is recorded as follows.

Metric	**Score**
Accuracy	0.954042259424542
Precision	0.9567170072056066
Recall	0.9515511486353819
f1 score	0.9541270856917853
roc_auc score	0.954053674084868

Table 11. Performance metrics on the trained model for predicting a medal winner pair between participation and athlete node pair for walk length 10, number of walks per each starting node 10

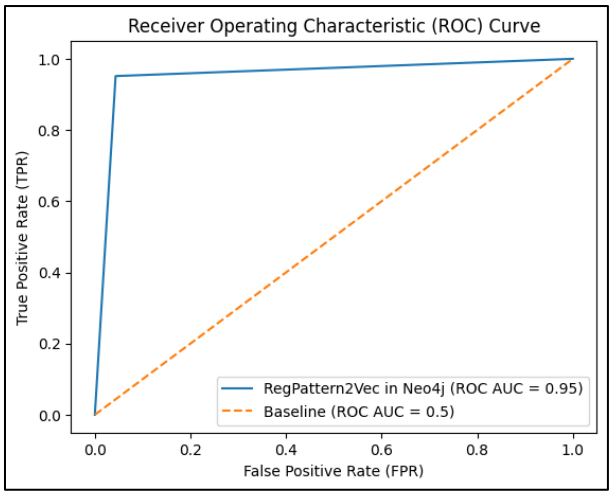


Figure 18. ROC plot of the Olympic Dataset experiment

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis, we presented a Neo4j implementation for a novel graph embedding algorithm RegPattern2Vec, where a regular expression's Deterministic Finite Automata guides a random walk to sample large Knowledge Graphs and learn node embeddings. This implementation has been observed to produce effective node embeddings and capture more contextual information of the Knowledge Graph without having to traverse the entire Knowledge Graph, except for the paths that satisfy the regular expression. We have tested this implementation on a citation network dataset and verified results through experiments and evaluations. This implementation can be directly applied to the data in the graph database Neo4j, avoiding the extra data preparation step compared to the original RegPattern2Vec implementation.

As a part of future work, we want to incorporate the conversion of regular expression to Deterministic Finite Automata within the plugin rather than manually writing it as string. This provides a benefit to directly provide the input in the Neo4j application as a regular expression instead of a string defined Deterministic Finite Automata of the regular expression in code. We also want to incorporate this plugin as an open-source application for the Neo4j team following their code guidelines and considerations.

REFERENCES

[1] Keshavarzi, A., Kannan, N., & Kochut, K. J. (2021). RegPattern2Vec: Link Prediction in Knowledge Graphs. In *2021 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*. https://doi.org/10.1109/iemtronics52119.2021.9422604

[2] *Graph Embeddings*. Neo4j. https://neo4j.com/developer/graph-data-science/graph-embeddings/

[3] *Graph embedding*. Snap-Stanford. https://snap-stanford.github.io/cs224w-notes/machine-learning-with-networks/node-representation-learning [Photograph]

[4] Neo4j. *Graph Data Science Library*. Neo4j. https://neo4j.com/docs/graph-data-science/current/

[5] Grover, A., & Leskovec, J. (2016). *node2vec*. https://doi.org/10.1145/2939672.2939754

[6] Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. In *Neural Information Processing Systems* (Vol. 30, pp. 1024–1034). https://doi.org/10.48550/arXiv.1706.02216

[7] Chen, H., Sultan, S. Z., Tian, Y., Chen, M., & Skiena, S. (2019). *Fast and Accurate Network Embeddings via Very Sparse Random Projection*. https://doi.org/10.1145/3357384.3357879

[8] Tan, Q., Liu, N., Zhao, X., Yang, H., Zhou, J., & Hu, X. (2020). *Learning to Hash with Graph Neural Networks for Recommender Systems*. https://doi.org/10.1145/3366423.3380266

[9] Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). *DeepWalk*. https://doi.org/10.1145/2623330.2623732

[10] Dong, Y., Chawla, N. V., & Swami, A. (2017). *metapath2vec*. https://doi.org/10.1145/3097983.3098036

[11] Church, K. (2016). Word2Vec. *Natural Language Engineering*, *23*(1), 155–162. https://doi.org/10.1017/s1351324916000334

[12] Richter, L. (2018). Jure Leskovec, AnandRajaraman, and Jeffrey D.Ullman. Mining of Massive Datasets. Cambridge, Cambridge University Press. *Biometrics*. https://doi.org/10.1111/biom.12982

[13] Mikolov, T., Chen, K., Corrado, G. S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. In *arXiv (Cornell University)*. Cornell University. https://arxiv.org/pdf/1301.3781

[14] Shi, C., Li, Y., Zhang, J., Sun, Y., & Yu, P. S. (2017). A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, *29*(1), 17–37. https://doi.org/10.1109/tkde.2016.2598561

[15] M. Spiser, Introduction to the Theory of Computation. Cengage learning., 2012

[16] JetBrains. *IntelliJ IDEA*. https://www.jetbrains.com/help/idea/discover-intellij-idea.html

[17] Github. *metapath2vec*. https://github.com/Change2vec/change2vec/tree/master/metapath2vec/code_metapath2vec

[18] animer. *DBLP Citation Network Dataset*. https://www.aminer.org/citation

[19] *Subset-dblp-data*. Google Drive. https://drive.google.com/drive/u/1/folders/1QXGFKaNsZTxKY2w_8YuxKBtYgzYitHC1

[20] Kaggle (n.d.). *120 Years of Olympic History: Athletes and results*. https://www.kaggle.com/datasets/heesoo37/120-years-of-olympic-history-athletes-and-results

[21] Neo4j. *APOC*. https://neo4j.com/docs/apoc/current/introduction/

[22] *Change2Vec*. Github. https://github.com/Change2vec/change2vec/tree/master/metapath2vec/code_metapath2vec

[23] *Scikit-learn*. Scikit-Learn.org. https://scikit-learn.org/stable/getting_started.html

[24] *NumPy*. Numpy.org. https://numpy.org/

[25] *Matplotlib*. Matplotlib. https://matplotlib.org/stable/users/release_notes.html#release-notes

[26] *Py2neo*. Py2neo. https://py2neo.org/2021.1/

[27] *Pandas*. PyData. https://pandas.pydata.org/about/index.html

[28] *PySpark*. Apache. https://spark.apache.org/docs/latest/api/python/

[29] *Plotly*. Matplotlib. https://matplotlib.org/stable/users/getting_started/

[30] *Jupyter*. Jupyter. https://docs.jupyter.org/en/latest/

[31] Principal Component Analysis. (2023, June 19). In *Wikipedia*.https://en.wikipedia.org/wiki/Principal_component_analysis

[32] Logistic Regression. (2023, May 30). In *Wikipedia*. https://en.wikipedia.org/wiki/Logistic_regression

[33] Chawla, N. V., Bowyer, K. W., Hall, L. J., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, *16*, 321–357. https://doi.org/10.1613/jair.953