**GRAPH EMBEDDING**

**Minor Project II**

Submitted by:

**Harsh Kumar Singh (19103184)**

**Sambbhav Jain (19103189)**

**Aryan Singh Tomar (19103200)**

Under the supervision of:

**DR. DHANALEKSHMI G.**

****

**Department of CSE/IT**

**Jaypee Institute of Information Technology University, Noida**

# **Month 2022**

**ACKNOWLEDGEMENT**

I would like to place on record my deep sense of gratitude to \_\_\_\_\_\_\_\_\_\_\_\_\_, Designation, Jaypee Institute of Information Technology, India for his/her generous guidance, help and useful suggestions.

I express my sincere gratitude to \_\_\_\_\_\_\_\_\_\_\_\_\_\_, Dept. of -----------------, India, for his/her stimulating guidance, continuous encouragement and supervision throughout the course of present work.

I also wish to extend my thanks to \_\_\_\_\_\_\_\_\_\_\_\_\_ and other classmates for their insightful comments and constructive suggestions to improve the quality of this project work.

**Signature(s) of Students**

HARSH KUMAR SINGH (19103184)

SAMBBHAV JAIN (19103189)

ARYAN SINGH TOMAR (19103200)

**Table of Contents**

.

***Page no .***

| *ACKNOWLEDGEMENT* | ***2*** |
| --- | --- |
| *DECLARATION* | ***4*** |
| *CERTIFICATE* | ***5*** |
| *ABSTRACT* | ***6*** |
| *INTRODUCTION* | ***7-11*** |
| *LITERATURE SURVEY* | ***11-15*** |
| *REQUIREMENT ANALYSIS* | ***16-20*** |
| *IMPLEMENTATION* | ***21-23*** |
| *RESULT AND DISCUSSION* | ***23-24*** |
| *CONCLUSION* | ***25*** |

**DECLARATION**

We hereby declare that this submission is our own work and that, to the best of our knowledge and beliefs, it contains no material previously published or written by another person nor material which has been accepted for the award of any other degree or diploma from a university or other institute of higher learning, except where due acknowledgment has been made in the text.

Place:

Date:

Name: Harsh Kumar Singh

Enrolment No: 19103184

Name: Sambbhav Jain

Enrolment No: 19103189

Name: Aryan Singh Tomar

Enrolment No: 1910200

**CERTIFICATE**

This is to certify that the work titled “Graph Embedding” submitted by Harsh Kumar Singh, Sambbhav Jain, Aryan Singh Tomar of B.Tech of Jaypee Institute of Information Technology, Noida has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of any other degree or diploma.

Digital Signature of Supervisor

Name of Supervisor

Designation

Date

**ABSTRACT**

Graphs are widely used in various real-world applications. Communication forums are large graphs of successive people, biologists using graphic communication graphs, while communication networks are graphs themselves. They use graphs of similarities in the field of text mining. The interest in making a learning machine on graphs is growing. They are trying to predict new friendships on social media, while biologists are predicting active protein labels. The mathematical operation of the graphs is limited and the use of machine specific learning methods in graphs is a challenge. In this case, embedding seems like a logical solution. Graph embeddings are the transformation of property graphs to a vector or a set of vectors. Embedding should capture the graph topology, vertex-to-vertex relationship, and other relevant information about graphs, subgraphs, and vertices. More properties embedder encode better results can be retrieved in later tasks.

**1.INTRODUCTION**

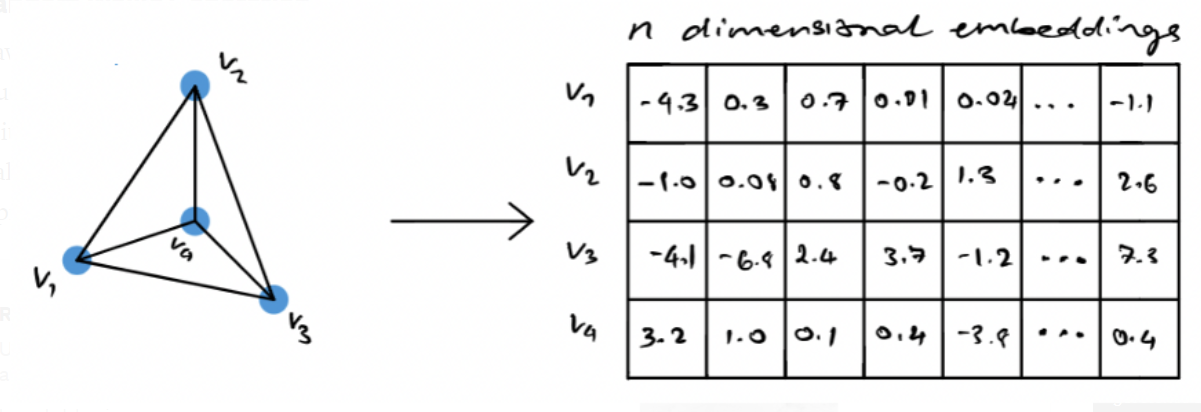
Now to start with this problem, Graphs are a meaningful and understandable representation of data, but there are a few reasons why graph embeddings are needed:

• **Machine learning on graphs is limited**. Graphs consist of edges and nodes. Those network relationships can only use a specific subset of mathematics, statistics, and machine learning, while vector spaces have a richer toolset of approaches.

• **Embeddings are compressed representations**. Adjacency matrix describes connections between nodes in the graph. It is a |V| x |V| matrix, where |V| is a number of nodes in the graph. Each column and each row in the matrix present a node. Non-zero values in the matrix indicate that two nodes are connected. Using an adjacency matrix as a feature space for large graphs is almost impossible. Imagine a graph with 1M nodes and an adjacency matrix of 1M x 1M. Embeddings are more practical than the adjacency matrix since they pack node properties in a vector with a smaller dimension.

• **Vector operations are simpler and faster** than comparable operations on graphs.

A notable problem when working with networks, is transforming the network structure into numerical representation which can then be passed onto traditional machine learning algorithms. Node2Vec is an algorithm that allows the user to map nodes in a graph G to an embedding space. Generally, the embedding space is of lower dimensions than the number of nodes in the original graph G. The algorithm tries to preserve the initial structure within the original graph. Essentially, the nodes which are similar within the graph will yield similar embeddings in the embedding space. These embedding spaces are essentially a vector corresponding to each node in the network. The graph embeddings are commonly used as input features to solve machine learning problems oriented around [link prediction](https://towardsdatascience.com/link-prediction-recommendation-engines-with-node2vec-c97c429351a8), community detection, classification, etc.



Generating n-dimensional node embeddings from a input graph G using node2vec (Image provided by author)

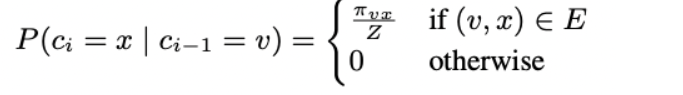
Generally, when dealing with very large graphs it’s quite difficult for scientists to visually represent the data they’re working with. A common solution to see how a graph look is to generate node embeddings associated with that graph and then visualize the embeddings in a lower-dimensional space. This allows you to visually see potential clusters or groups forming in very large networks.

**Random Walks Generation**

Having an understanding of what random walks are and how they work is crucial in understanding how node2vec works. I’ll provide a high level overview of it, but if you want a more intuitive understand and implementation of random walks in python you can read the article I’ve previously written regarding this topic.

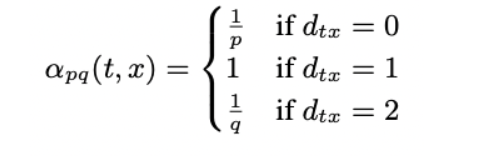
As a high level overview, the simplest comparison of a random walk would be through walking. Imagine, that each step you take is determined probabilistically. This implies that at each index of time, you have moved in a certain direction based on a probabilistic outcome. This algorithm explores the relationship to each step that you would take and its distance from the initial starting point.

Now you might wonder how these probabilities of moving from one node to another is calculated. Node2Vec introduces the following formula for determining the probability of moving to the node x given that you were previously at the node v.

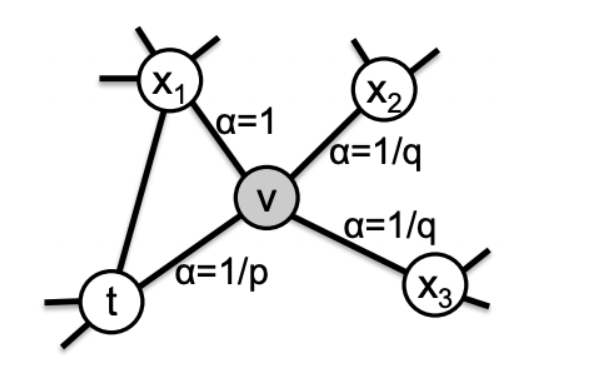


Where z is the normalization constant, and πvx is the unnormalized transition probability between nodes x and v [4]. Clearly, if there is no edge connecting x and v, then the probability will be 0, but if there is an edge, we identify a normalized probability of going from v to x.

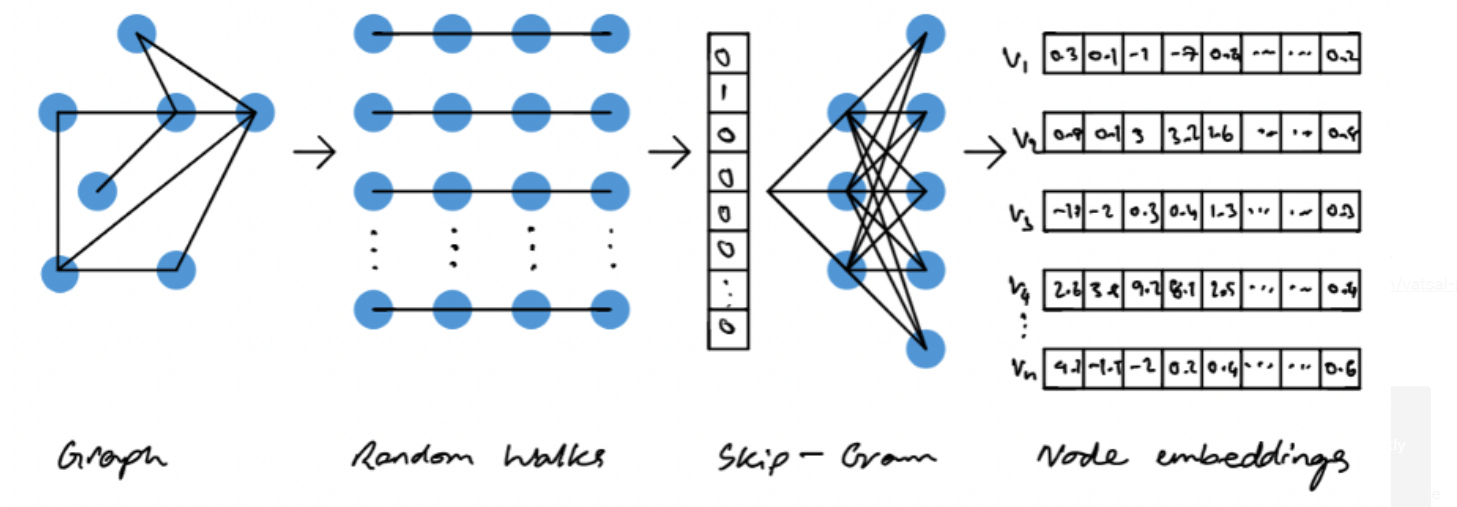
The paper states that the easiest way to introduce a bias to influence the random walks would be if there was a weight associated to each edge. However, that wouldn’t work in the case of unweighted networks. To resolve this, the authors introduced a guided random walk governed by two parameters p and q. p indicates the probability of a random walk getting back to the previous node, and q indicates the probability that a random walk can pass through a previously unseen part of the graph [4].



Where dtx represents the shortest path between nodes t and x. It can be visually seen in the illustration below.



The process for node2vec is fairly simple, it begins by inputting a graph and extracting a set of random walks from the input graph. The walks can then be represented as directed sequence of words where each node represents a word. The generated random walks are the passed into the skip-gram model. As explained above, the skip-gram model works on words and sentences, each node in the random walk can be represented as a word and the entire walk can be represented as a sentence. The result of the skip-gram model yields an embedding for each node (or word in this analogy). The entire process can be seen below.



**2. LITERATURE SURVEY**

Dealing with relational data always required significant computational resources, domain expertise and task-dependent feature engineering to incorporate structural information into a predictive model. Nowadays, a family of automated graph feature engineering techniques has been proposed in different streams of literature. So-called graph embeddings provide a powerful tool to construct vectorized feature spaces for graphs and their components, such as nodes, edges and subgraphs under preserving inner graph properties. Using the constructed feature spaces, many machine learning problems on graphs can be solved via standard frameworks suitable for vectorized feature representation. Our survey aims to describe the core concepts of graph embeddings and provide several taxonomies for their description. First, we start with the methodological approach and extract three types of graph embedding models based on matrix factorization, random-walks and deep learning approaches. Next, we describe how different types of networks impact the ability of models to incorporate structural and attributed data into a unified embedding. Going further, we perform a thorough evaluation of graph embedding applications to machine learning problems on graphs, among which are node classification, link prediction, clustering, visualization, compression, and a family of the whole graph embedding algorithms suitable for graph classification, similarity and alignment problems. Finally, we overview the existing applications of graph embeddings to computer science domains, formulate open problems and provide experiment results, explaining how different networks properties result in graph embeddings quality in the four classic machine learning problems on graphs, such as node classification, link prediction, clustering and graph visualization. As a result, our survey covers a new rapidly growing field of network feature engineering, presents an in-depth analysis of models based on network types, and overviews a wide range of applications to machine learning problems on graphs.

Keywords: Graph embedding, Knowledge representation, Machine learning, Network science, Geometric deep learning, Graph neural networks, Node classification, Link prediction, Node clustering, Graph visualization

**2.1 DATASET DESCRIPTION**

We demonstrate the node2vec technique on the [small version of the Movielens dataset](https://files.grouplens.org/datasets/movielens/ml-latest-small-README.html) to learn movie embeddings. Such a dataset can be represented as a graph by treating the movies as nodes, and creating edges between movies that have similar ratings by the users. The learnt movie embeddings can be used for tasks such as movie recommendation, or movie genres prediction.

MovieLens dataset includes around 100k ratings from 610 users on 9,742 movies. Dataset contains three files: movies.csv, ratings.csv, users.csv.

This dataset (ml-latest-small) describes 5-star rating and free-text tagging activity from [MovieLens](http://movielens.org/), a movie recommendation service. It contains 100836 ratings and 3683 tag applications across 9742 movies. These data were created by 610 users between March 29, 1996 and September 24, 2018. This dataset was generated on September 26, 2018.

Users were selected at random for inclusion. All selected users had rated at least 20 movies. No demographic information is included. Each user is represented by an id, and no other information is provided.

The data are contained in the files links.csv, movies.csv, ratings.csv and tags.csv. More details about the contents and use of all these files follows.

This is a *development* dataset. As such, it may change over time and is not an appropriate dataset for shared research results.

# **Content and Use of Files**

## **Formatting and Encoding**

The dataset files are written as [comma-separated values](http://en.wikipedia.org/wiki/Comma-separated_values) files with a single header row. Columns that contain commas (,) are escaped using double-quotes ("). These files are encoded as UTF-8. If accented characters in movie titles or tag values (e.g. Misérables, Les (1995)) display incorrectly, make sure that any program reading the data, such as a text editor, terminal, or script, is configured for UTF-8.

## **User Ids**

MovieLens users were selected at random for inclusion. Their ids have been anonymized. User ids are consistent between ratings.csv and tags.csv (i.e., the same id refers to the same user across the two files).

## 

## **Movie Ids**

Only movies with at least one rating or tag are included in the dataset. These movie ids are consistent with those used on the MovieLens web site (e.g., id 1 corresponds to the URL <https://movielens.org/movies/1>). Movie ids are consistent between ratings.csv, tags.csv, movies.csv, and links.csv (i.e., the same id refers to the same movie across these four data files).

## **Ratings Data File Structure (ratings.csv)**

All ratings are contained in the file ratings.csv. Each line of this file after the header row represents one rating of one movie by one user, and has the following format:

userId,movieId,rating,timestamp

The lines within this file are ordered first by userId, then, within user, by movieId.

Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars).

Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

## **Tags Data File Structure (tags.csv)**

All tags are contained in the file tags.csv. Each line of this file after the header row represents one tag applied to one movie by one user, and has the following format:

userId,movieId,tag,timestamp

The lines within this file are ordered first by userId, then, within user, by movieId.

Tags are user-generated metadata about movies. Each tag is typically a single word or short phrase. The meaning, value, and purpose of a particular tag is determined by each user.

Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

## **Movies Data File Structure (movies.csv)**

Movie information is contained in the file movies.csv. Each line of this file after the header row represents one movie, and has the following format:

movieId,title,genres

Movie titles are entered manually or imported from <https://www.themoviedb.org/>, and include the year of release in parentheses. Errors and inconsistencies may exist in these titles.

Genres are a pipe-separated list, and are selected from the following:

Action, Adventure, Animation, Children's, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western.

## **Links Data File Structure (links.csv)**

Identifiers that can be used to link to other sources of movie data are contained in the file links.csv. Each line of this file after the header row represents one movie, and has the following format:

movieId,imdbId,tmdbId

**3. REQUIREMENT ANALYSIS**

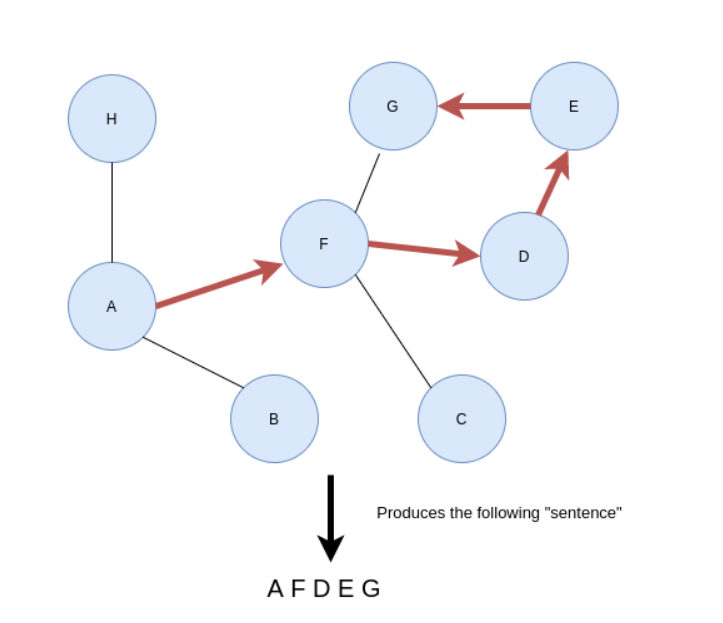
**3.1 ALGORITHMS USED**

1. **NODE2VEC ALGORITM**

## **Node2vec**

Node2vec algorithm uses skip-gram with negative sampling (SGNS). But how do we create a text corpus as input to the SGNS? Node2vec uses random walks to generate a corpus of “sentences” from a given network.

A random walk can be interpreted as a drunk person traversing the graph. Of course, you can never be sure of an intoxicated person’s next step, but one thing is sure. A drunk person traversing the graph can only hop onto a neighboring node.

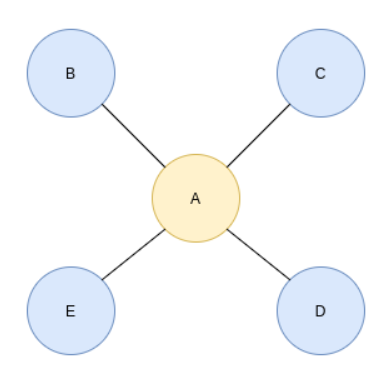


Construct sentences based on random walks.

Imagine if you start traversing the graph from node A. At random, you pick a neighboring node and hop on to it. Then you repeat the process until a predefined **walk length**. The walk length parameter defines how long the “sentences” will be. For every node in the graph, the node2vec algorithm generates a series of random walks with the particular node as the starting node. You can define how many random walks should start from a particular node with the **walks per node** parameter. To sum it up, the node2vec algorithm uses random walks to generate a number of sentences starting from each node in the graph. The walk length parameter controls the length of the sentence. Once the sentences are generated using random walks, the algorithm inputs them into the SGNS model and retrieve the hidden layer weights as node embeddings. That is the whole gist of the node2vec algorithm.

However, the node2vec algorithm implements second-order biased random walks. A step in the first-order random walk only depends on its current state.

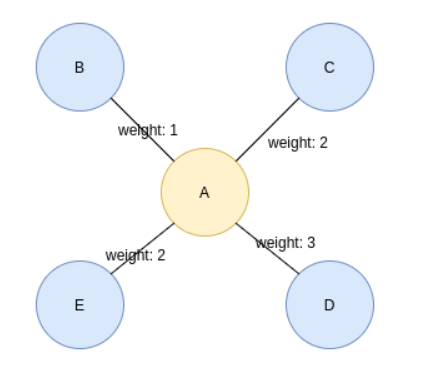
## **First-order random walks**



First-order random walk. Image by the author.

Imagine you have somehow wound up at node A. Because the first-order random walk only looks at its current state, the algorithm doesn’t know which node it was at the earlier step. Therefore, the probability of returning to a previous node or any other node is equal. There is no advanced math concept behind the calculation of probability. Node A has four neighbors, so the chance of traversing to any of them is 25% (1/4).

Suppose your graph is weighted, meaning that each relationship has a property that stores its **weight**. In that case, those weights will be included in the calculation of the traversal probability.

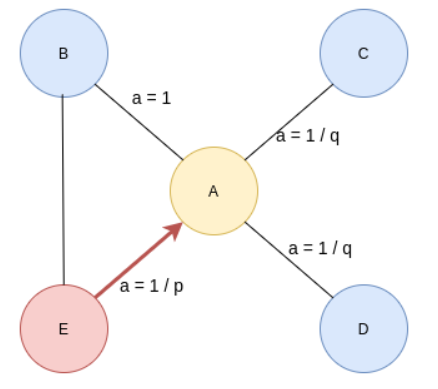


First-order random walk on a weighted graph. Image by the author.

In a weighted graph, the chance of traversing a particular connection is its weight divided by the sum of all neighboring weights. For example, the probability to traverse from node A to node E is 2 divided by 8 (25%) and the probability to traverse from node A to node D is 37.5%.

## **Second-order biased random walks**

Second-order walks take into account both the current as well as the previous state. To put it simply, when the algorithm calculates the traversal probabilities, it also considers where it was at the previous step.



Second-order biased random walk. Image by the author.

The walk just traversed from node E to node A in the previous step and is now evaluating its next move. The likelihood of backtracking the walk and immediately revisiting a node in the walk is controlled by the **return** parameter p. Setting a high value to parameter p ensures lower chances of revisiting a node and avoids 2-hop redundancy in sampling. This strategy also encourages moderate graph exploration. On the other hand, if the value of the p parameter is low, the chances of backtracking in the walk are higher, keeping the random walk closer to the starting node.

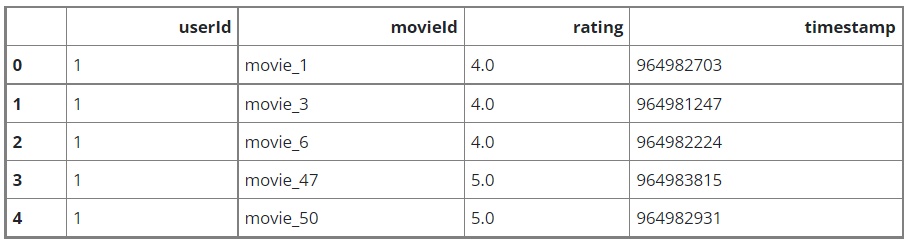
The **inOut** parameter q allows the traversal calculation to differentiate between inward and outward nodes. Setting a high value to parameter q (q > 1) biases the random walk to move towards nodes close to the node in the previous step. Going to the previous image, if you set a high value for parameter q, the random walk from node A is biased towards nodes closer to node E. Such walks obtain a local view of the underlying graph with respect to the starting node in the walk and approximate breadth-first search. In contrast, if the value of q is low (q < 1), the walk is more inclined to visit nodes further away from node E. This strategy encourages outward exploration and approximates depth-first search.

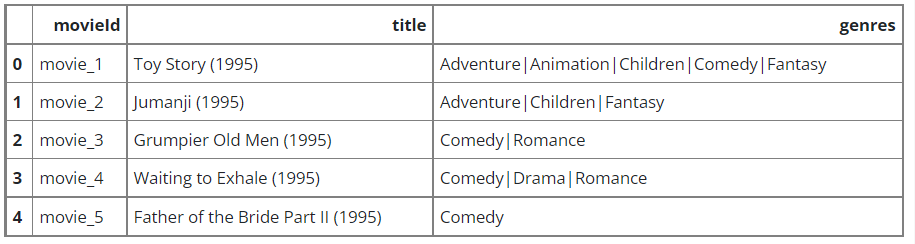
**4. IMPLEMENTATION**

**4.1 DATA PREPARATION**

We load the data into a Pandas DataFrame and perform some basic preprocessing :-

* Load movies to a DataFrame.
* Create a `movieId` string.
* Load ratings to a DataFrame.
* Convert the `ratings` to floating point
* Create the `movie\_id` string.

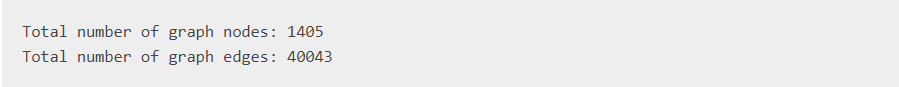




**4.2 ESTABLISHING RELATIONSHIPS BETWEEN MOVIES**

We create an edge between two movie nodes in the graph if both movies are rated by the same user >= min\_rating. The weight of the edge will be based on the [pointwise mutual information](https://en.wikipedia.org/wiki/Pointwise_mutual_information) between the two movies, which is computed as: log(xy) - log(x) - log(y) + log(D), where:

* xy is how many users rated both movie x and movie y with >= min\_rating.
* x is how many users rated movie x >= min\_rating.
* y is how many users rated movie y >= min\_rating.
* D total number of movie ratings >= min\_rating.

****

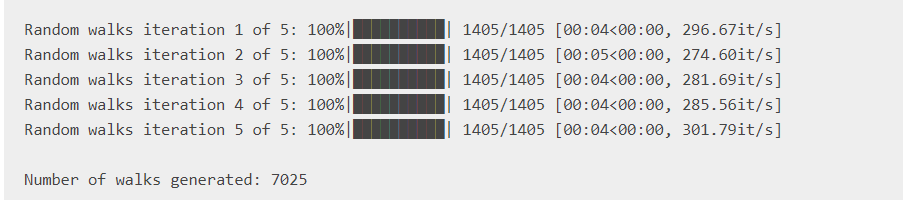
**4.3 IMPLEMENTING BIASED RANDOM WALK**

## 

A random walk starts from a given node, and randomly picks a neighbour node to move to. If the edges are weighted, the neighbour is selected *probabilistically* with respect to weights of the edges between the current node and its neighbours. This procedure is repeated for num\_steps to generate a sequence of *related* nodes.

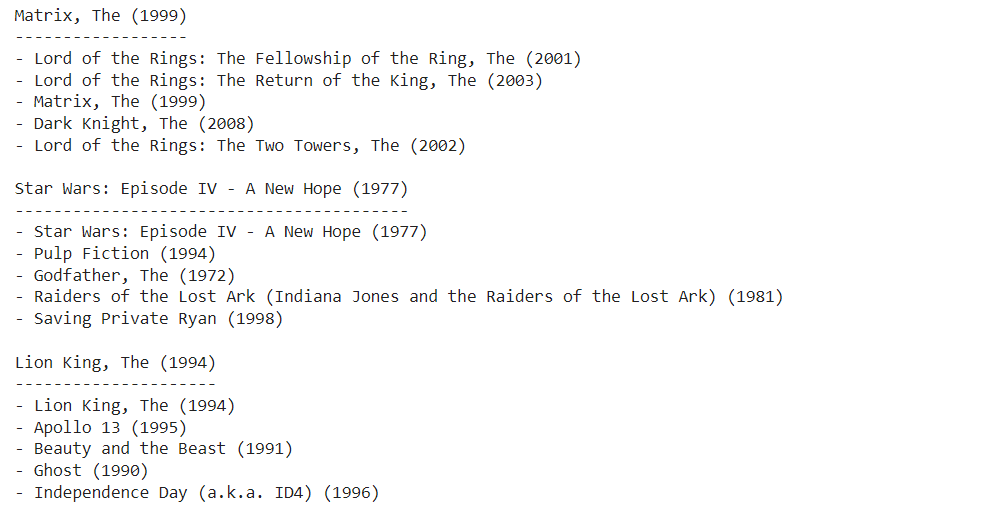
The [*biased* random walk](https://en.wikipedia.org/wiki/Biased_random_walk_on_a_graph) balances between breadth-first sampling (where only local neighbours are visited) and depth-first sampling (where distant neighbours are visited) by introducing the following two parameters:

1. Return parameter (p): Controls the likelihood of immediately revisiting a node in the walk. Setting it to a high value encourages moderate exploration, while setting it to a low value would keep the walk local.
2. In-out parameter (q): Allows the search to differentiate between *inward* and *outward* nodes. Setting it to a high value biases the random walk towards local nodes, while setting it to a low value biases the walk to visit nodes which are further away.

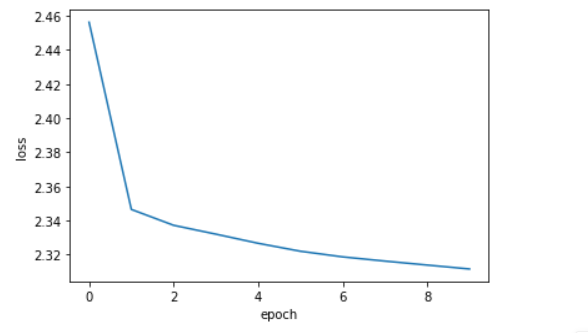


**5. RESULTS AND DISCUSSION**

Results given by our embeddings:-



Our graph returned similar movies based on their genres, ratings etc. This concludes that movies with same genre are closely packed together.



(Loss Vs Epoch graph)

Shape of Embeddings:-



**6. CONCLUSION**

We demonstrate the node2vec technique on the [small version of the Movielens dataset](https://files.grouplens.org/datasets/movielens/ml-latest-small-README.html) to learn movie embeddings. Such a dataset can be represented as a graph by treating the movies as nodes, and creating edges between movies that have similar ratings by the users. The learnt movie embeddings can be used for tasks such as movie recommendation, or movie genres prediction.We train our model using node2vec technique and we succeeded in successfully embedding of graph and generate related movies according to user’s input.

**7.REFERENCES**

1. [**https://towardsdatascience.com/**](https://towardsdatascience.com/)
2. **X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang, “Community preserving network embedding,” in AAAI, 2017, pp. 203–209.**
3. [**https://keras.io/examples/graph/node2vec\_movielens/**](https://keras.io/examples/graph/node2vec_movielens/)
4. **N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, “Navigating the maze of graph analytics frameworks using massive graph datasets,” in SIGMOD, 2014, pp. 979–990**
5. **K. M. Borgwardt and H. Kriegel, “Shortest-path kernels on graphs,” in ICDM, 2005, pp. 74–81**
6. **https://snap.stanford.edu/node2vec/**