JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Enes Sovtic

Angefertigt am
Institut für Computergrafik

Erstbetreuer
Univ.-Prof. Dr. Marc Streit

Zweitbetreuer
Christian Steinparz, MSc

September 2023

Interactive Graph Node Positioning via Machine Learning Embeddings

Bachelorarbeit
zur Erlangung des akademischen Grads
Bachelor of Science
im Bachelorstudium
Artificial Intelligence

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, September 2023 Enes Sovtic

Abstract

Graph Layouting describes the process of creating a visual representation of a node-link-graph on a 2D plane. Over the years, two distinct classes of layouting algorithms have emerged: Topology- or structure driven methods utilize the connections between nodes to calculate their positions. Such approaches excel in aesthetic criteria like minimizing edge crossings and node occlusions. A newer class of algorithms depends on node attributes to generate a layout. These are called attribute-driven methods and they often consist of machine learning tools, specifically embedding algorithms. Those show good results in highlighting community information. Inspired by the work of Shen et al., 2023 [28], this bachelor thesis aims at combining these two approaches and developing a layout algorithm that encompasses the merits of both. The goal is to provide an interface that lets the user smoothly transition between the two. This documentation will explain the process of integrating these two classes of algorithms, how their combined advantages and drawbacks influence the results, and under which circumstances such an application works best.

Kurzfassung

Graphzeichnen oder Graph Layouting bezeichnet das Erstellen einer visuellen Realisierung eines Node-Link-Graphen auf einer zweidimensionalen Ebene. Über die Jahre hinweg haben sich zwei unterschiedliche Klassen an Layoutalgorithmen aus dem Themengebiet herausgebildet: Strukturbasierte Methoden berechnen die Knotenpositionen anhand der dazwischenliegenden Kanten. Solche erzielen gute Resultate in ästhetischen Gesichtspunkten, wie die Anzahl der Kantenkreuzungen oder Knotenüberlappungen zu minimieren. Attributbasierte Methoden nutzen die Knotenattribute um ein Layout zu generieren. Dabei handelt es sich oft um Machine Learning Algorithmen, im Speziellen Embedding-Algorithmen. Diese sind besonders gut darin, Cluster erkennbar zu machen. Durch die Arbeit von Shen et al., 2023 inspiriert [28], zielt diese Bachelorarbeit darauf ab, diese beiden Ansätze zu kombinieren und einen Layoutalgorithmus zu entwickeln, der die Vorzüge beider demonstriert. Für die Anwendung soll eine Benutzeroberfläche erstellt werden, mit derer sich das Layout dynamisch anpassen lässt, indem zwischen den beiden Ansätzen gleitend interpoliert wird. Diese Dokumentation beschreibt den Prozess der Integration dieser beiden Klassen von Algorithmen, wie deren kombinierte Vor- und Nachteile die Ergebnisse beeinflussen und unter welchen Umständen solch eine Anwendung am besten funktioniert.

Contents

List of Figures

List of Tables

1 Introduction

Graph drawing problems or graph layout problems describe the task of finding a placement of the nodes and edges in a graph on a plane that fulfills desired criteria, like optimising an objective function or creating an aesthetically pleasing visualization for inspecting the contained information. To this end, a plethora of automatic graph drawing algorithms have been researched and developed over the years [3].

Most classical algorithms only utilize the connections between the nodes to generate a layout. The arguably most popular class are force-directed algorithms. Those simulate the graph as a physical system. Nodes repel each other unless they are connected by an edge, in which case they get pulled together. The strength of the repelling and attracting forces depends primarily on the distance between the nodes. The nodes keep pulling and repelling each other until, ideally, after a certain number of iterations all movement stops and a stable layout is achieved [7].

Nowadays, graph theory differentiates between many types of graphs [33], some of which fall under the term "attributed". Graphs in this class not only give information about the type and strength of connections between nodes, but also about further individual characteristics of every single node in tabular form (e.g. a social graph represents people as nodes and lists age, gender etc. as attributes).

Since the rise of machine learning, efforts have been made to utilize dimensionality reduction methods for visualizing graph data [10]. The node connections or attributes or both are embedded and used to create high-dimensional representations, which are then embedded onto a 2D-plane, ultimately forming the graph layout.

The results generated by force-directed and dimensionality reduction methods show some unique characteristics. The former are quite intuitive and easily implementable. They tend to be unsuitable for larger graphs where the visualization just ends up looking like a so-called "hairball". The placement of and the distance between nodes are only influenced by the connections and other nodes between them. Attributes do not influence the placement, such that nodes with similar features are not necessarily close together.

The latter yield good results in identifying neighborhoods of similar nodes. Depending on the algorithm and its parameter settings, this effect may often lead to the graph becoming overly compact as the nodes are grouped together too closely. Additionally, if the edge information is badly or not at all encoded into the embeddings, connected nodes are not guaranteed to be close together, which leads to the edges connecting all over the place from one end to the other. This occludes most of the graph, defeating the purpose of the visualization.

There have been attempts at combining the two approaches but their results often tend to focus more on one of the methods over the other [13], [29], [2]. The motivation behind this bachelor thesis is to build an algorithm that combines both methods and encompasses both their strengths. This algorithm shall then be used in an application that lets the user generate a layout for a given graph by smoothly transitioning between the force-directed and the embedding approach based on how much weight they want to give to each of the two.

2 Theoretical Background

This section explains the most important terms and background information needed for the understanding of the algorithms described in this document. This includes a general introduction to graph theory, a further in-depth description of the two types of layouting methods and a discussion about machine learning embeddings.

2.1 General Graph Theory

Graph theory is a research subject focused on the study of graphs. In the mathematical sense, a graph is described as a construct that models pairwise relations between objects [34]. In notation it is written as a pair $G = (V, E)$, where:

- V is a finite set, called the vertices (nodes) of G, and
- E is a subset of $\mathcal{P}_2(V)$ (i.e. a subset of all possible two-element combinations of V), called the edges (arcs) of G.

An alternative definition describes a graph as a set of triples. A triple has the form $t = (s, p, o)$, where

- $s \in V$, the subject, which is a given node in the graph
- $o \in V$, the object, which is another node in the graph
- $p \in E$, the predicate, which is an edge connecting the subject and object.

This definition is used in the Resource Description Framework (RDF) data model to codify a directed (see 2.1.1), semantic statement. An example would be $(Human, drives, Car)$, which is represented as a node "Human" connecting to a node "Car" via an edge named "drives". The concept of triples and the RDF model are especially important in regard to Knowledge Graphs, which semantically store information as statements in a graph structure.

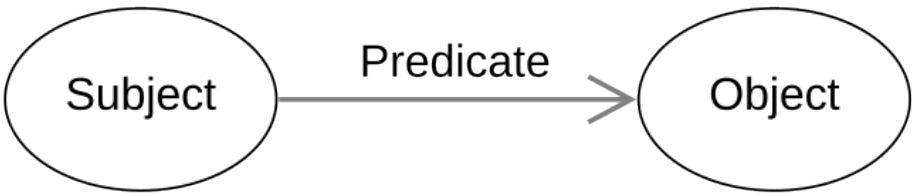


Fig. 2.1: RDF triple

These two definitions describe the most basic structure of a graph. Depending on certain characteristics, many more different types of graphs have been formalized over the years. The most relevant ones for this thesis will be described here in short.

It should be mentioned that term "network" is sometimes used in the same context as "graph" in the literature. For the purpose of this thesis, only the term "graph" will be used for the associated mathematical structure.

2.1.1 Un-/Directed Graphs

Edges in a graph can be directed. That means that the relation from one node to another is not symmetric:

$$(s, p, o) \not\Longrightarrow (o, p, s) \tag{2.1}$$

On the other hand, an undirected graph has no directed edges and therefore for all relations the following applies:

$$(s, p, o) \Longrightarrow (o, p, s) \tag{2.2}$$

2.1.2 Dis-/Connected Graphs

In a connected graph there exists for every possible pair of nodes a path made of edges such that every node can be visited by every other node going along the edges between them. On the other hand, in a disconnected graph there exists at least one pair of nodes such that no path between them can be found by following along the edges to other nodes.

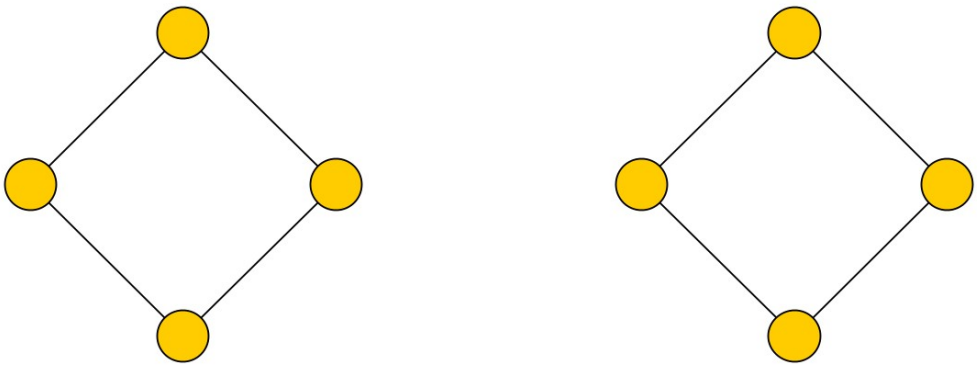


Fig. 2.2: Disconnected graph

2.1.3 Attributed Graph

Additionally to the sets V and E containing the nodes the nodes and edges respectively, an attributed graph also contains the set of all attribute vectors $\Lambda = \{\alpha_1, ..., \alpha_m\}$ associated with the nodes in V. Each α_i is of size $|V|$ and contains the corresponding attribute value for every node. Each node v_i in V then has a vector $(\alpha_1(v_i), ..., \alpha_m(v_i))$ for its attributes.

An attributed graph therefore has the form $G = (V, E, \Lambda)$.

2.2 Graph Layouting

The process of creating a visual representation of a graph on a (usually 2D) plane is called Graph Layouting or Graph Drawing. The goal is to create a depiction of the graph structure that is aesthetically pleasing, optimizes certain objectives, fulfills given constraints and does so in the most computationally efficient way possible.

The drawing of the graph should not be confused with the graph itself. The totality of all nodes and edges in the mathematical structure of a given graph is unambiguous, while there are virtually infinite ways to represent that same graph visually.

Depending on the desired properties of the visualization, many different suitable methods have been researched over the years [3]. The purpose of this thesis is to explore and combine two distinct classes of layouting algorithms, which primarily differ in the way they utilize the graph data. Both of these draw strictly straight edges, so their only objective is to find positions for the nodes.

2.2.1 Topology-driven Layout Methods

Topology driven layout methods use the edge information of the graph to generate positions for the nodes. The most prominent example are force directed layout algorithms. They spatialize a graph representation by simulating physical forces and applying them to the nodes.

Nodes generally repel each other while edges attract their nodes. How exactly those forces are calculated varies depending on the algorithm. Though nowadays, most of them use the same set of functions for it and replace some of the parameters according to the desired behaviour [16]:

$$F_a = k * d^a \qquad\qquad F_r = -k * d^r \tag{2.3}$$

F_a	force of attraction
F_r	force of repulsion
d	distance between nodes
a	attraction parameter; $a \in \mathbb{Z} \geq 0$
r	repulsion parameter; $r \in \mathbb{Z} < 0$
k	predetermined constant

The strength of the repelling and attracting forces is affected primarily by the distance between the nodes (for attraction proportionally, for repulsion inverse proportionally). Depending on the parameters for attraction and repulsion, the forces can be set to scale linearly or quadratically with the distance (other parameter values are usually not used [14]).

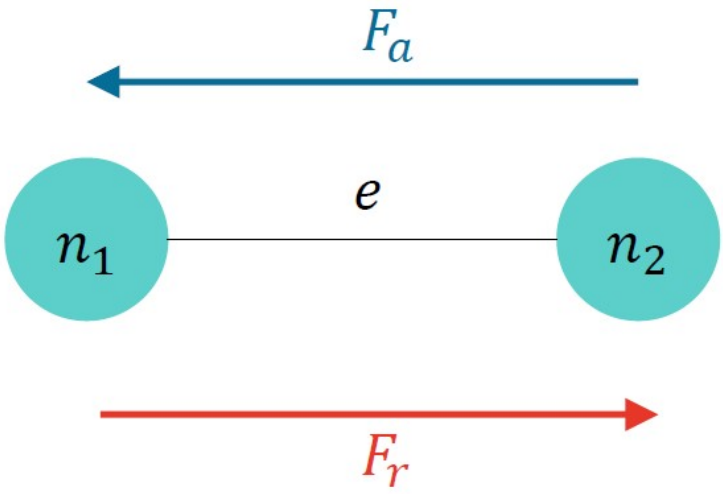


Fig. 2.3: Attraction-repulsion model of force-directed algorithms

Different calculation methods are uniquely described by the parameters a and r, which is why the (attraction, repulsion)-model with the shorthand notation (a, r) has been developed to describe the parameters of different algorithms. The most popular force-directed layout algorithm, the Fruchterman-Reingold algorithm [7], has the model (2, -1) (quadratic in attraction, linear in repulsion).

Starting from an initial position, the forces are calculated in every iteration and added up to modify the node positions. Eventually the algorithm converges and all forces sum to zero, producing a stable layout.

The F-R algorithm typically generates aesthetically pleasing layouts that reduce edge crossings and node occlusions. Its implementation is easy and intuitive. Its drawbacks are that the results depend on the initial state and are therefore non-deterministic. The process can get stuck in local minima. It also may not converge at all, which is why a maximum number of iterations should be set. The coordinates of a single node does not hold any meaning. They can only be evaluated relative to others.

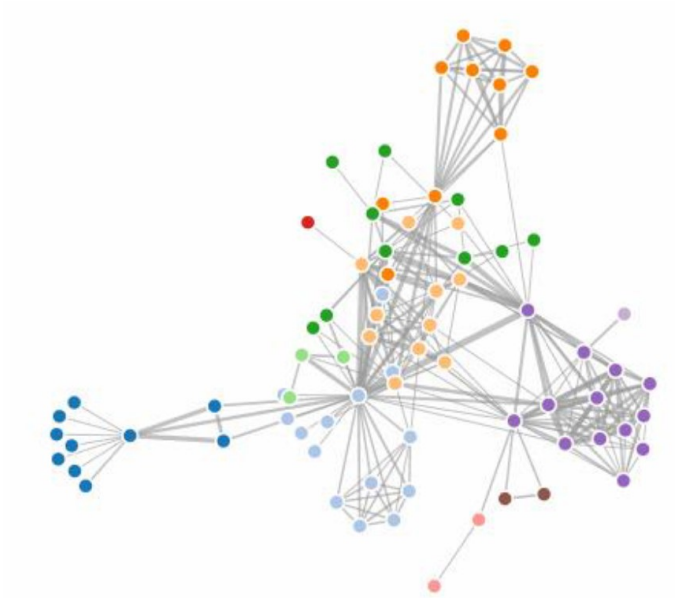


Fig. 2.4: Force-directed methods implicitly reduce edge crossings, node occlusions.

2.2.2 Attribute-driven layout methods

Algorithms in this class of layout methods do not encode the topological structure of the graph. Instead they use the attribute vectors associated with the nodes and downproject them to two-dimensional embeddings, which can be used to position the corresponding nodes. Since those are available in tabular form, machine learning approaches are well suited for that task.

Multiple different technologies have been tested and applied for the purpose of this thesis. The following sections will explain them in more detail.

t-SNE

t-distributed stochastic neighbor embedding (t-SNE) is one of the most popular dimensionality reduction techniques used in machine learning [20]. It is an improvement of stochastic neighbor embedding (SNE) [12], which computes the pairwise probabilities that a data point x_i would choose a node x_j as its neighbor in proportion to their probability density under a Gaussian centered at x_i. Nearby points have a high probability, separated points have a low one. These probabilities are computed both for the high- ($p_{j|i}$) and the low dimensional ($q_{j|i}$) space. SNE tries to match $q_{j|i}$ to $p_{j|i}$.

This is done via gradient descent. A large issue of SNE is its cost function, which is difficult to optimize. t-SNE replaces that cost function with one that has simpler gradients, so that it is less prone to falling into local minima. Further, it uses a t-distribution rather than a Gaussian to compute the probabilities in the low-dimensional space.

t-SNE was, at its publication, vastly more powerful than similar dimensionality reduction methods and is to this day widely in use due to its performance and its simplicity in regard to implementation and parameter choice, which is why it was incorporated in this project.

UMAP

Uniform Manifold Approximation and Projection (UMAP) is a dimensionality reduction method similar to t-SNE [21]. According the publishers it "is competitive with t-SNE for visualization quality and arguably preserves more of the global structure with superior run time performance".

Compared to t-SNE, the implementation of UMAP uses much more advanced mathematics, which will be explained here in a strongly simplified way. At their basis, the algorithms are quite similar. Both create a high- and low-dimensional representation of the data and try to match them. In UMAP, those representations are actually graph structures with edge weights indicating the probability of two nodes being connected. Those are determined by building a radius around each point just long enough to connect to the other nearest radius.

UMAP is a novel approach compared to t-SNE, which significantly improves upon it by applying a multitude of mathematical theorems that guarantee a good result under fast performance.

3 Related Work

This thesis was strongly inspired by a paper from Tsinghua University, Beijing, China titled "Graph Exploration with Embedding-Guided Layouts" written by Leixian Shen, Zhiwei Tai, Enya Shen, and Jianmin Wang [28].

The authors explore the currently available graph layout methods and implement a pipeline that creates low-dimensional embedding vectors from the node attributes and feeds them to a force-directed layout algorithm to produce a visualization that combines the advantages of both approaches.

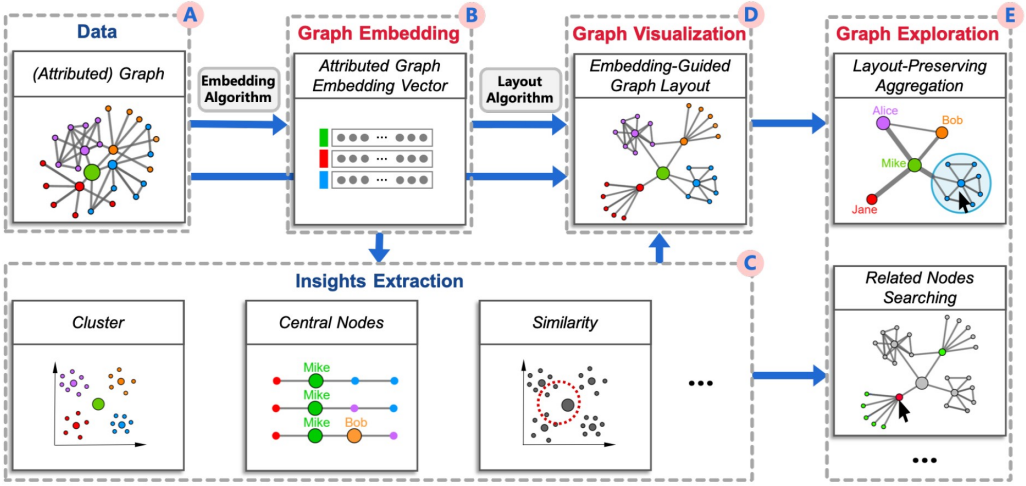


Fig. 3.1: Graph visualization and exploration pipeline from Shen et al., 2023

Citing the paper: "Embedding-based graph exploration pipeline. The graph embedding algorithm encodes attributed graphs into low-dimensional vectors, from which can extract rich data insights (e.g., cluster, central nodes, and similarity. The embedding-guided layout method integrates similarities and connections between nodes to produce aesthetic and community-aware layouts. The generated layout, coupled with the extracted insights, enables various graph exploration applications (e.g., node aggregation and related nodes searching)".

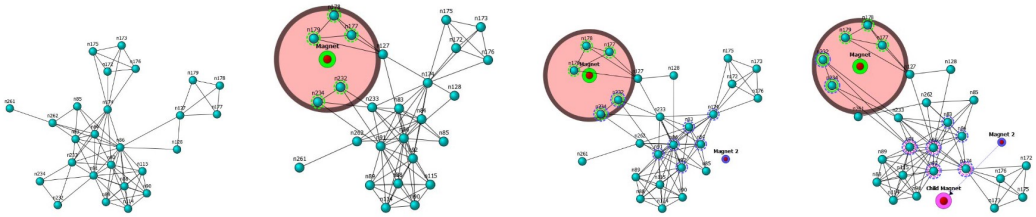
Many of the techniques used in the paper were adapted for this thesis. Details about their implementation will be discussed in the next chapter.

Another related project is "Design and Evaluation of MagnetViz—A Graph Visualization Tool" by Andre Suslik Spritzer and Carla Maria Dal Sasso Freitas [30].

The core of their work is an interactive application named MagnetViz, which allows its user to generate a graph using a force-directed algorithm and then to alter that layout according to their preferences. The tool enables the user to place so-called virtual magnets inside the graph structure.

Those magnets can be configured to attract certain nodes based on user-defined criteria such as attribute-based characteristics like having a given attribute value or topology-based features like their degree or the number of adjacent nodes. Nodes can even be selected based on their relation to other magnets.

Once a magnet is placed, the force-directed layout starts to rearrange itself "in order to reflect the changes in the balance of forces, consequently changing the visualization into one that is more semantically relevant to the user".



(a) Original layout. (b) A simple attraction. (c) Magnet intersection. (d) Magnet hierarchy.

Fig. 3.2: (Cited from paper) Using the MagnetViz tools on a small graph.

Another similar approach, which is also mentioned in Shen et al., 2023 [28], is GraphTSNE [18]. The developers of that algorithm incorporate information from both node features and graph structure by utilizing Graph Convolutional Networks (GCN) with a modified tSNE-loss. By altering a hyperparameter, the layout, specifically the degree of class separation, can be modified.

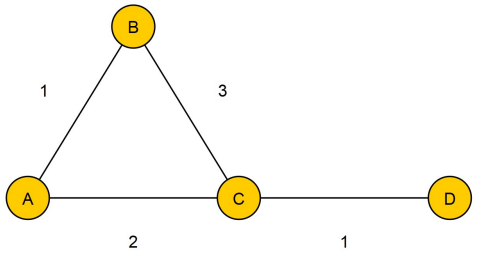
4 Approach

The sections in this chapter will describe the components needed for the application and the steps taken to build and implement them. A number of different versions of the program were developed based on which algorithms are used for a selection of the components. The performances of those different approaches are compared later in the chapter "Results".

The application will be referred to as IGL-algorithm (**I**nteractive **G**raph **L**ayout) in this and the following chapters to make a clearer distinction between the product of this thesis and other mentioned approaches.

4.1 Basic idea

A typical graph can also be represented as an adjacency-matrix, a $|V| * |V|$-matrix, where both the rows and columns stand for all the nodes in the graph. The cell values contain the strength of the connection between the nodes given by the corresponding row and column.



	A	B	C	D
A	0	1	2	0
B	1	0	3	0
C	2	3	0	1
D	0	0	1	0

Fig. 4.1: Graph in topological and matrix representation. An undirected graph produces a diagonal matrix.

Many graph positioning algorithms (e.g. force-directed) accept this matrix form to calculate the layout of the graph. It has the advantage that it allows to efficiently set the strength of connection, i.e. the "closeness" in the visualization between every pair of nodes.

The basic idea behind the implementation is to create two of these matrices: one containing the topological information (edge weights), from here on called the adjacency matrix, and one containing the attribute information (distance between embedding vectors of corresponding nodes), from here on called the similarity matrix. Their weighted sum forms the final adjacency-matrix, which is then fed to a force-directed algorithm to generate the positions for the visualization. The calculation goes as follows:

$$N = wA + (1 - w)S \tag{4.1}$$

N final adjacency matrix for the visualization
A adjacency matrix extracted from the graph topology
S similarity matrix extracted from the node embedding vectors
w parameter weighting the influence of the two matrices; $w \in [0, 1]$

Depending on the weight applied to those two matrices the output can either appear more similar to a force-directed approach or to a machine learning embedding approach. The user can then freely configure that parameter and therefore the visualization depending on their preferences.

4.2 Input

The IGL-algorithm was designed for the computation and visualization of attributed graphs. For it to function it needs two sets of data:

- A table containing the information for the node attributes. Every row represents a node in the graph. The first column in every row should hold the ID of every node. Subsequent columns hold the values for the attributes. Optionally, in data sets that have a class label for the nodes, the corresponding column should be titled "label".

- A table containing the edge information. Every row represents an edge and each row has three columns: The source node, the target node and the edge weight.

It should be mentioned that in the current version of the algorithm, all edge weights are set to 1, regardless of the input. This is because having a vast range of different weights led to irregular behaviour during experiments. Applying this solution produced better results. Therefore, the values in that column can be chosen arbitrarily for the input.

4.3 Embedding vectors

This section explains how the two-dimensional embedding vectors are created. The high-dimensional node attribute vectors are input into the down projection algorithm to generate the embeddings.

For this thesis, three different down projection algorithms were used in the experiments. The first two, t-SNE and UMAP, were introduced in the theoretical background chapter. The third method is called node2vec and was used in the paper mentioned in the related work chapter with promising results [28]. This section will give an introduction to it.

4.3.1 node2vec

node2vec is a feature learning algorithm specifically developed for the representation of nodes and their neighborhoods in a graph [8]. According to the authors, the approach "returns feature representations that maximize the likelihood of preserving network neighborhoods of nodes in a d-dimensional feature space".

The algorithm can be divided into two phases: random walk simulation and embedding generation with word2vec. In the first phase, the algorithm picks a node in the graph at random. From there, it starts randomly visiting adjacent nodes for a defined number of steps. At the end, it saves all the visited nodes as a consecutive sequence, hereby referred to as a "walk". Then it starts again from another random node. This keeps going until it has a certain number of walks, which are then used to generate the embeddings.

The behaviour regarding which nodes are visited during the walks depends on the parameters p and q. In the following schematic, the walk just visited node v coming from node t. It has three different possible choices from there on with three different normalized probabilities α:

- Returning to the previous node t with $\alpha = \frac{1}{p}$
- Visiting a previously seen node x_1 with $\alpha = 1$
- Going to a node from an, as of yet, unseen part of the graph x_2 or x_3 with $\alpha = \frac{1}{q}$

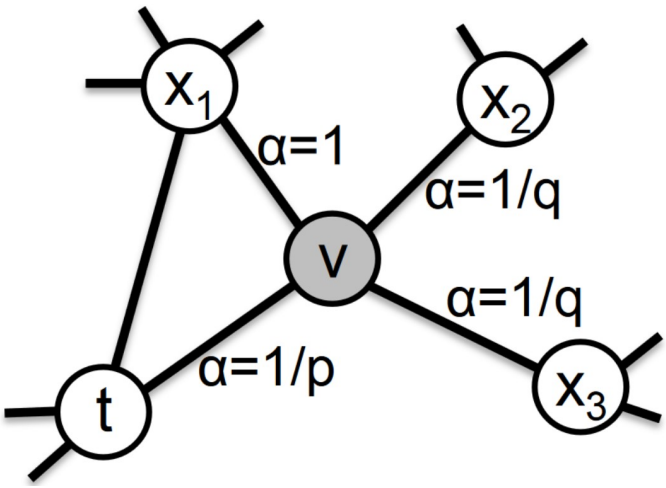


Fig. 4.2: Single step of the random walk generation process [8]

Formally, let c_i denote the ith node in the walk, starting with $c_0 = u$. Nodes c_i are then generated by the following probability distribution:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

where π_{vx} is the unnormalized transition probability between nodes v and x, and Z is the normalizing constant. In the given case where all edge weights have the value 1, the unnormalized transition probability equals the transition probability $\pi_{vx} = \alpha_{vx}$ with:

$$\alpha_{vx} = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases} \tag{4.3}$$

where $d_{tx} \in \{1, 2, 3\}$ is the shortest distance between the previous node and the next node in the walk.

By adjusting the parameters p and q the random walk behaviour can be influenced. With a lower p, the individual random walks explore the local structure and close neighborhood of the starting node. A higher value encourages outward exploration into global structure. For q the opposite is the case. The parameter choice reflects directly in the vector representations, such that the user can choose to let the embeddings emphasize local or global structure.

The next few paragraphs will give a high-level overview of word2vec, which is necessary to understand how the embeddings are generated from the walks. Word2vec utilizes a skip-gram model architecture to generate representations for words in a corpus [23]. In the training process, the model takes a sentence from the corpus and selects a target word. It then tries to predict the words surrounding it in the sentence by calculating probabilities for every possible word that could be in the target word's context.

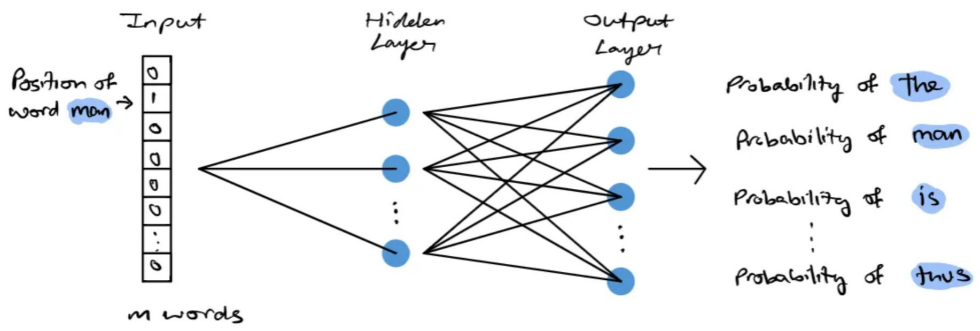


Fig. 4.3: Skip-Gram Model architecture [26]

This training process is conducted for all words and sentences in the corpus until, ideally, the learned embeddings of the words in vector space yield the maximum accuracy on the test set. Words that often share a context should be close together while words that form an uncommon combination are distant to each other.

Node2vec utilizes the same architecture but the words are replaced by nodes and the sentences by walks. In each walk, the model uses one of the nodes to predict the surrounding nodes. At the end of training, the model has produced representations of the nodes in vector space, such that those which often appear in the same walk and close together are close in vector space. Nodes that seldom appear in the same walk because they are separated by many steps in the graph have distant embeddings.

Node2vec gives the user a lot of freedom in deciding what kind of connections should be represented in the embeddings. Despite that, the standard model does not process node attribute information and it is therefore not encoded into the embeddings, ultimately not influencing the visualization. The authors of the paper mentioned in "Related Work" [28] present an adapted version of node2vec that solves that problem.

For every appearing value of every attribute in the nodes, a virtual node is placed into the graph. Every real node that has that value for the corresponding attribute is connected to it via a virtual edge. These nodes can then be visited in the random walk procedure. For that, an additional parameter r is added to the process that decides the next step in the walk.

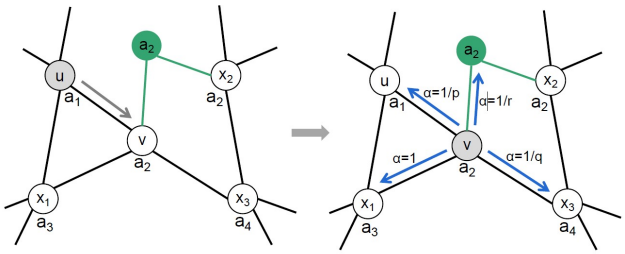


Fig. 4.4: Extension of node2vec that models attribute information [28]

Let V_Λ denote the set of all virtual attribute nodes in the graph. The transition probability for that version of the algorithm, titled node2vec-a, is designed as:

$$\pi_{vx} = \begin{cases} \frac{1}{r} & \text{if } v \text{ or } x \in V_\Lambda \\ \alpha_{vx} & \text{otherwise} \end{cases} \tag{4.4}$$

The probability of visiting an attribute node is given by the parameter r. For all other adjacent nodes the behaviour is the same as in the standard algorithm. By altering the three parameters the resulting embeddings can be influenced to emphasize on reflecting local structure, global structure or node attributes.

4.4 Similarity matrix

This section describes the creation of the similarity matrix for the embeddings. By calculating the pairwise euclidean distances between all node vectors we get the distance matrix D, which is then normalized into the range [0,1]. We can think of the distance matrix as the opposite of the similarity matrix, so for the last step we have:

$$S = 1 - D \tag{4.5}$$

S normalized similarity matrix
D normalized distance matrix extracted from the node embedding vectors

The result is a diagonal $|V| * |V|$-matrix that shows the similarity measure between the attribute vectors of every possible node pair. Naturally, the similarity between a node and itself is maximal, so the diagonal is filled with 1's. These values are replaced with 0's to not imply self-connections in the final graph. The matrix should then look similar to this:

	A	B	C	..
A	0	0.4	0.7	..
B	0.4	0	0.95	..
C	0.7	0.95	0	..
..	..	..	..	..

Table 4.1: Similarity matrix

4.5 Adjacency matrix

The adjacency matrix contains information on the pairwise topological connections between all nodes. This information is simply extracted from the set of all edges. The result is a $|V| * |V|$-matrix filled with 0's and 1's similar to this one:

	A	B	C	..
A	0	0	1	..
B	0	0	1	..
C	1	1	0	..
..	..	..	..	..

Table 4.2: Adjacency matrix

4.6 Final adjacency matrix

This section describes the procedure of combining the similarity- and adjacency matrices created in the two previous sections into the final adjacency matrix and the computation steps following it. As mentioned in the beginning of the chapter, the final matrix is computed according to this formula:

$$N = wA + (1 - w)S \tag{4.6}$$

- N final adjacency matrix for the visualization
- A adjacency matrix extracted from the graph topology
- S similarity matrix extracted from the node embedding vectors
- w parameter weighting the influence of the two matrices; $w \in [0, 1]$

The weighting parameter is supposed to be set by the user in the IGL-algorithm. The visualization should update after a parameter change with no noticeable delay. Since the computation of the final adjacency matrix and the resulting position based on it takes a considerable amount of time, it was decided that those steps should not be done during runtime but at the start of the algorithm.

To this end, n linearly spaced weight values in the range of [0,1] are hard encoded into the algorithm. Once the program is started, a different version of the final adjacency matrix is computed for each value of w. The result looks similar to table 4.1. At this stage, most likely every cell of the matrix will be filled with the majority of the values being rather low. This is expected, since most node pairs should have low similarity except some relatively few with a higher similarity.

To amplify the distance between dissimilar nodes, a truncation operation was implemented at this step. Similarity values below a defined threshold are set to zero.

$$t(x) = \begin{cases} 0 & x < t_e \\ x & \text{otherwise} \end{cases} \tag{4.7}$$

A slightly enhanced truncation function is used for datasets that have a distinguished label for every node. It makes sense to group nodes with the same label together and to distance those from different clusters from each other. That is why a lower threshold is applied for in-group nodes than out-group ones:

$$t'(x, u, v) = \begin{cases} 0 & x < p_t(u, v) \\ x & \text{otherwise} \end{cases} \tag{4.8}$$

where p_t is a function that determines the value of the threshold:

$$p_t(u,v) = \begin{cases} t_{in} & \text{if label}(u) = \text{label}(v) \\ t_{out} & \text{if label}(u) \neq \text{label}(v) \end{cases} \tag{4.9}$$

Each of these n truncated adjacency matrices is then input into a conventional Fruchterman-Reingold algorithm. Finally, the resulting sets of coordinates are then used to draw the graph. Once the user changes the weight parameter, a different set of coordinates is applied and the graph is re-rendered with no discernible delay.

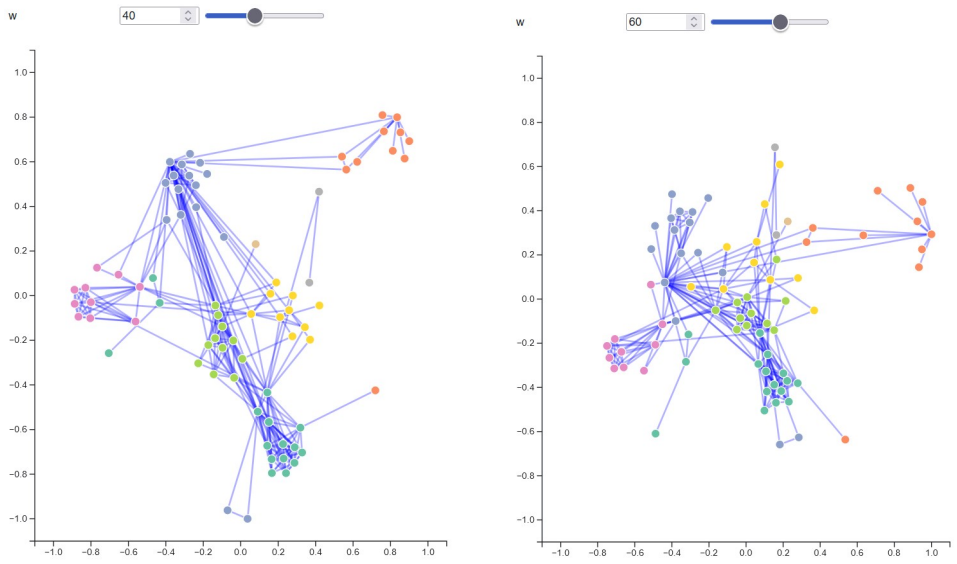


Fig. 4.5: Working example of the IGL-algorithm adjustable by a slider

5 Implementation

This chapter will describe the technical details of the implementation. This includes the used software packages and algorithms and their configuration.

5.1 Software

5.1.1 Programming Language

All software that was produced over the course of this thesis was written using the Python programming language since it allows the easy integration of a vast library of software packages, including tools for the creation, manipulation and analysis of graph structures [32].

5.1.2 Software Packages

All used packages were integrated via Conda, "an open source package management system and environment management system" [1]. It offers a simple interface for the installation, editing and deletion of Python packages and other languages. The software can be organized in environments, which allows for a clear separation between projects and also makes version control much easier.

The experiments were conducted in Jupyter notebooks, a web-based development environment that allows for quick testing of code, analysis of data and arranging workflows [15].

Numpy is a staple Python package nowadays, which extends the language's functionality with the ndarray datatype, which allows for faster numerical computation and comprehension compared to Python's rather slow native lists [11]. This package was used for all the mathematical operations on the adjacency-/similarity matrices.

The node- and edge information used as the input for the IGL-algorithm were formatted as dataframes included in the Pandas package [22]. It is among the most popular packages for data analysis and manipulation. The dataframe format allowed for easy preparation of the input data.

The embedding algorithm tSNE was used via its implementation in scikit-learn, a popular software library containing a plethora of tools for machine learning [27]. The UMAP algorithm has its own package created by the inventors of the method [31]. Node2vec was also included through a dedicated package [25]. As mentioned in the previous chapter, the algorithm had to be adapted by adding the third probability r for walking to adjacent virtual nodes.

The needed tools for creating, modifying and visualizing graphs was provided by the NetworkX package [9]. It offers an interface for creating the graph structure from both the edges dataframe and the final adjacency matrix. Additionally, it implements the Fruchterman-Reingold algorithm and allows the user to calculate the node positions based on it given a graph structure.

NetworkX also has a visualization feature. While it is completely satisfactory for testing and generating results, it does have a slight delay when the weight parameter is changed, which is disadvantageous when trying to view the changes between different configurations. To that end, the positions computed in Python were exported to an Observable notebook. Observable is an online platform that primarily aims at making quick visualizations in an user-friendly environment [4]. It is similar to Jupyter but uses Javascript instead of Python. The interface in Observable can be seen in figure 4.5.

5.2 Parameter Choice

For most of the algorithms, the parameters were set to default. The embedding algorithms were configured to downproject to two dimensions. Node2vec has a lot more parameters that have to be configured manually. The number of random walks that should be generated was set to 150. The walk length was 30. The walk strategy parameters were configured as $p = 1, q = 0.8$ and $r = 0.7$ to maintain a slight bias towards attribute information. The truncation thresholds for graphs with labels were configured as $t_{in} = 0.4$ and $t_{out} = 0.6$ to encourage clusters with the same labels. For datasets without label the threshold is set to 0.5.

6 Evaluation

This chapter describes the steps taken to measure the performance of the IGL-algorithm and the results of those measurements. The performance will be judged both quantitatively, on the basis of a selection of defined metrics, and qualitatively, by visual comparison. The tests will be conducted at different weight settings on a number of datasets. Additionally, the results will be compared to some baselines. The discussion of the results will be carried out in the next chapter.

6.1 Datasets

The algorithm is designed for attributed graph datasets, of which fewer are freely available compared to graphs without attributes. For the evaluation, the following three datasets were picked:

Name	# nodes	# edges	# attributes	Label	Description
Les Miserables[17]	77	254	0	Yes	Characters graph of Victor Hugo's novel.
Facebook[19]	61	540	48	No	Social graph from Facebook.
Webk(Cornell)[5]	183	298	1703	Yes	Webpage citation graph of Cornell University

Table 6.1: List of datasets used for evaluation

The first dataset is a graph structure containing co-occurrences of characters in Victor Hugo's novel 'Les Misérables'. A node represents a character and an edge between two nodes shows that these two characters appeared in the same chapter of the the book. The weight of each link states how often they appeared together. The labels were defined by M. E. J. Newman and M. Girvan using an algorithm for discovering community clusters, as described in his paper "Finding and evaluating community structure in networks" [24]. The labels are also the only feature of the nodes in this dataset.

The second graph is taken from a dataset consisting of "circles" (or "friends lists") from Facebook. It contains information on ten different sets of people who are directly or indirectly connected via the "friends" feature, which the edges represent. The node attributes consist of information found in the users' account page, although anonymized. Each circle is stored as a distinct graph in the dataset, one of which was used for this thesis.

Webk is a dataset that includes web pages from computer science departments of selected universities, where each university is represented by its own graph. This thesis uses the Cornell subset, where nodes represent web pages and edges are links between them. Node features are the bag-of-words representation of the text content of the pages. The pages/nodes are classified into five labels: student, project, course, staff, and faculty.

6.2 Procedure

Evaluating the performance of the IGL-algorithm is not straightforward, since the output is dynamically changed by the weight parameter, depending on the preferences of the user regarding aesthetics and certain metrics. This is why four different, fixed values of the parameter will be taken as representative configurations to judge the behaviour of the application.

These are set as $w \in \{10, 30, 50, 70\}$. Experiments have shown that no large changes are observed for $w > 70$. This is most likely because the similarities only influenced by the embeddings are then so low that they get truncated. Left are only the similarities given by the edge information, which are always $w*1$ or higher. This creates a layout that is almost identical to a pure force-directed layout. Additionally, the tests will be run for three different embedding algorithms to compare their performances as well (tSNE, UMAP, node2vec).

For WebK, the number of attributes for the node2vec approach are reduced to 50 via tSNE. The reason for that is explained in the Discussions chapter.

The results are compared against three different baselines, both qualitatively and quantitatively. The first baseline is a Fruchterman-Reingold algorithm like it is used for this thesis. The other two baselines are approaches with a similar idea as IGL but with much simpler execution.

The first one entails creating embeddings from the node features and then feeding those as initial coordinates to a force-directed algorithm with 50 iterations (from here on referred to as embed $\rightarrow$ force). The second approach is the same but in reverse (force $\rightarrow$ embed). The chosen embedding algorithm is TSNE with 1000 iterations.

7 Results

7.1 Qualitative Results

7.1.1 Les Miserables

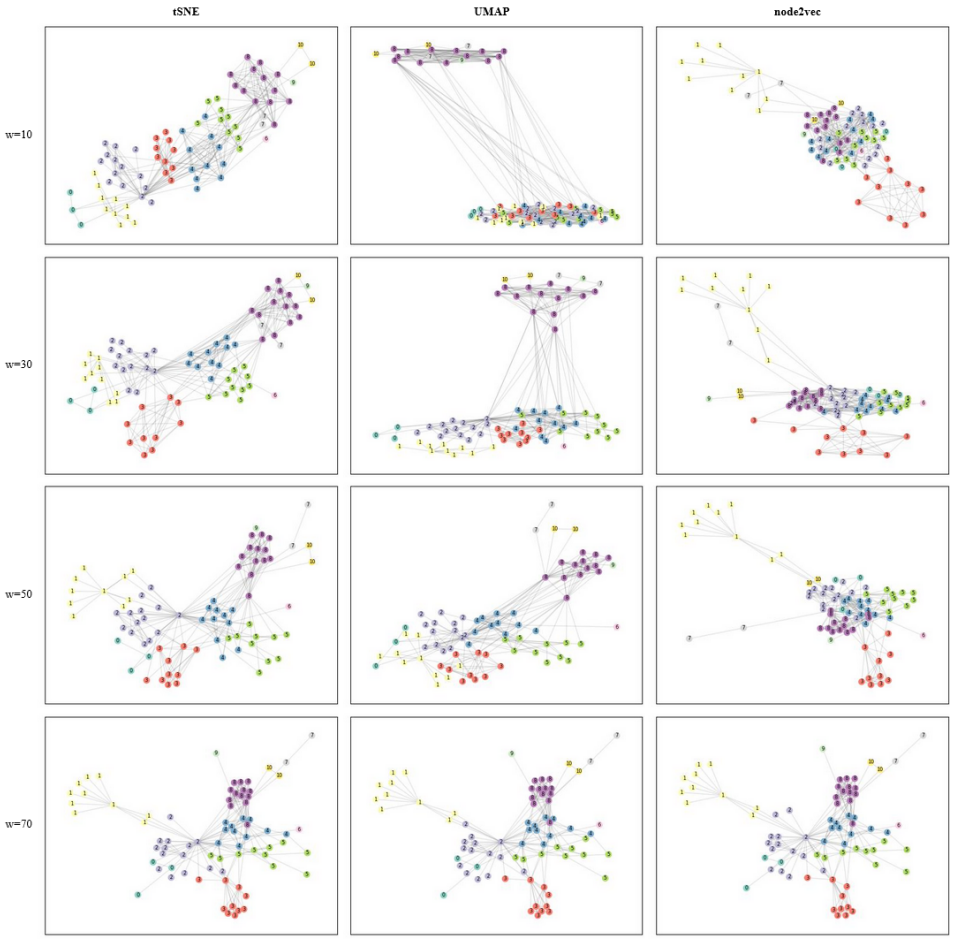


Fig. 7.1: Results of the IGL-algorithm for Les Miserables

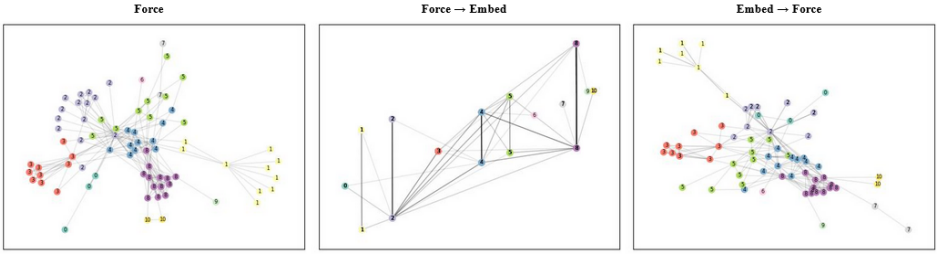


Fig. 7.2: Results of the baselines for Les Miserables

7.1.2 Facebook

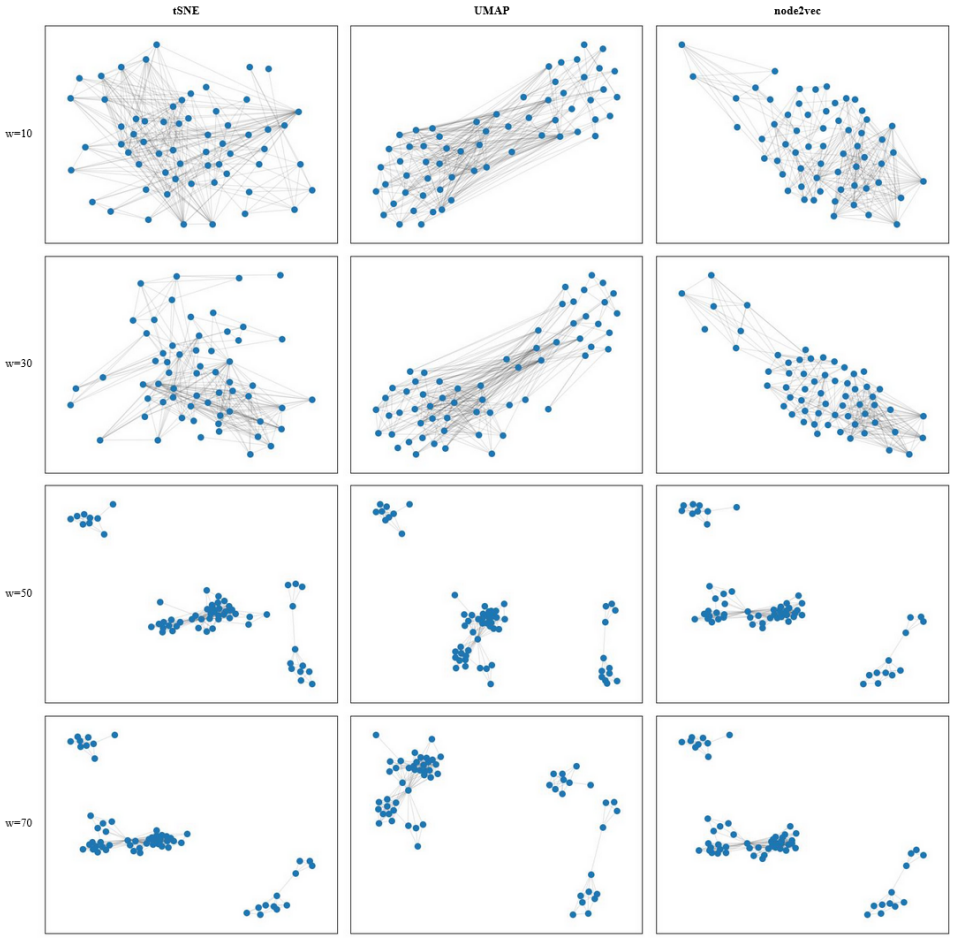


Fig. 7.3: Results of the IGL-algorithm for Facebook

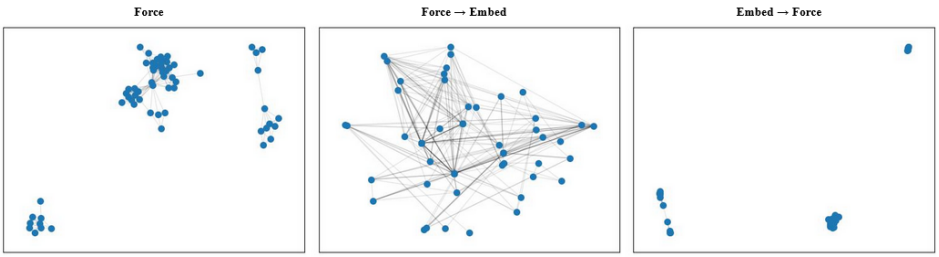


Fig. 7.4: Results of the baselines for Facebook

7.1.3 WebK

Fig. 7.5: Results of the IGL-algorithm for WebK

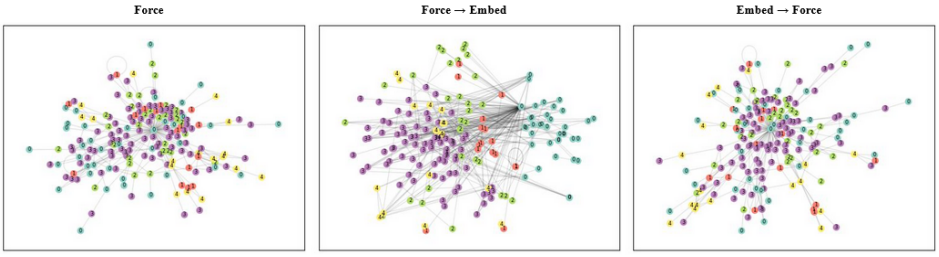


Fig. 7.6: Results of the baselines for WebK

7.2 Quantitative Results

7.2.1 Metrics

The performance of graph layout algorithms is computationally evaluated on defined metrics that are based on factors that influence the readability of the resulting layout [6]. For the purpose of this thesis, the most popular ones have been adapted:

Edge crossings

Overlapping edges can make it difficult to follow the connections from and to nodes and therefore negatively influence readability of the graph. Computationally, the number of edge crossings is determined by checking every possible edge pair and counting those that cross. To allow comparison between graphs of different sizes, that number is further normalized by the total number of possible edge pairs. To sum up, the metric E_c shows the percentage of edge pairs that cross:

$$E_c = \frac{|\{(e_1, e_2) \in E \times E \mid cross(e_1, e_2)\}|}{|E \times E|} \tag{7.1}$$

E_c	Edge crossing metric $\in [0, 1]$
$E \times E$	Set of all edge pairs

This metric is quite expensive to compute since every possible edge pair has to be checked. Since no edge has to be checked against itself and no pair has to be checked twice, this results in the complexity $O((n-1)\frac{n}{2})$.

Node occlusions

At the end of the layout process it may happen that nodes are placed at the same coordinates or very close to each other, which affects readability negatively. The number of node occlusions is determined by counting the node pairs whose distances between the two nodes lie under a certain threshold. This number is then again divided by the total number of node pairs, resulting in the metric N_{oc}.

$$N_{oc} = \frac{|\{(v_1, v_2) \in V \times V \mid occlude(v_1, v_2)\}|}{|V \times V|} \tag{7.2}$$

N_{oc}	Node occlusion metric $\in [0, 1]$
$V \times V$	Set of all node pairs

Computationally, the number of node occlusions is determined in the same way as the number of edge crossings, which results in the algorithm having the same worst case computational complexity. The average case can be greatly reduced though, by sorting the set of all nodes by e.g. the x-coordinate. The location of the first node is then compared to all the following ones. Once the distance on the x-axis exceeds the threshold, the loop can be stopped, since the distance can only get greater than the threshold, and the comparisons from the second node are started.

7.2.2 Results Table

		Les Miserables		Facebook		WebK	
		E_c	N_{oc}	E_c	N_oc	E_c	N_oc
tSNE	w=10	0.030	**0.000**	0.077	**0.000**	0.087	0.002
	w=30	0.019	**0.000**	0.052	**0.000**	0.076	0.002
	w=50	0.014	0.001	0.023	0.014	0.041	0.004
	w=70	0.014	0.002	0.022	0.019	0.005	0.002
UMAP	w=10	0.048	0.071	0.077	**0.000**	0.105	0.003
	w=30	0.027	0.011	0.070	**0.000**	0.107	0.005
	w=50	0.019	0.002	0.021	0.013	0.045	0.001
	w=70	0.015	0.001	0.022	0.008	0.005	0.002
node2vec	w=10	0.049	0.001	0.046	**0.000**	0.099	0.056
	w=30	0.036	0.025	0.037	**0.000**	0.072	0.027
	w=50	0.021	0.004	0.020	0.017	0.066	0.008
	w=70	0.014	0.002	0.020	0.018	0.005	0.002
baselines	FR	0.016	0.001	**0.019**	0.012	**0.004**	0.002
	force→embed	0.025	0.033	0.071	0.012	0.126	**0.000**
	embed→force	**0.013**	0.006	0.036	0.212	0.005	0.003

Table 7.1: Quantitative results of the experiments on the datasets.

Table 7.1 shows the results of the quantitative measurements of the IGL-algorithm for all weights and embedding algorithms for all datasets and of all baselines likewise. The lowest/best values of each column are written in bold and the highest/worst values of each column are written in red. The discussion of the results can be found in the next chapter.

8 Discussion

The datasets that were selected to evaluate the performance of the IGL-algorithm have some distinguished characteristics. While all three of them have similar sizes in regard to topological attributes like the number of nodes and edges, they do vary concerning their attribute information.

The first dataset, Les Miserables, has only one attribute, which simultaneously acts as the label for the nodes regarding group-/cluster membership. The Facebook dataset has a moderate number of attributes but none that acts as a label. WebK is again a labeled dataset with an extraordinary number of attributes compared to the topological graph size.

Looking at the outputs generated by the experiments it becomes clear that Facebook also has, aside from the other features, a disconnected graph structure. That fact is not easily recognizable without drawing a visualization of the connections, which leads to the first observation about the algorithm's behaviour.

The final step of the IGL-pipeline is a force-directed algorithm, which, by definition, repels nodes from each other unless they are connected by an edge. If the algorithm is applied to a dataset like Facebook, the disconnected pieces of the graph shoot farther and farther away from each other with each iteration.

This behaviour can be observed in the experiments in figure 7.3. In IGL, this effect can be mitigated by setting a lower weight, where the layout is influenced mostly by the outputs generated by the embedding algorithms. Continuously increasing the weight leads to visualization becoming more and more force-directed-like.

The greater the distances between the individual pieces of the graph become, the smaller appear the distances between the connected nodes contained in those pieces. At some point they become hardly distinguishable from each other, which is reflected in the high node occlusion value for greater values of w in the results table 7.1.

A similar phenomenon appears in the visual outputs of the node2vec algorithm for the WebK-dataset in figure 7.5 at weights 10 and 30. Most of the nodes in the graph concentrated at one position of plot in one big mass, which reflects in the node occlusion metric, except for some singular nodes, which are placed at opposite ends of the visualization.

The cause of that behaviour can be traced back to the truncation step described in section 4.6. Sometimes it may happen that, for a given node, the similarity values to all other nodes in the final adjacency matrix are set to zero because they all happen to be lower than the configured threshold.

The matrix is then fed to a force-directed algorithm, which leads to the disconnected node being shot away from the other parts of the graph, as described before, even though it is connected by an edge in the actual graph structure. This error can be mitigated by setting a lower threshold at the expense of resulting in a weaker distinction between dissimilar nodes.

Looking at the results generally, the expected behaviour can be observed: for lower values of w, the metrics, which are based on topological features, are worse than for higher values. The opposite is true in regards to the formation of distinct clusters. Nodes from the same group tend to lie in the same areas.

Comparing the performance of the different embedding methods regarding the output, it can be observed that tSNE seems to have the most predictable results, largely displaying the expected behaviour described in the previous paragraph. UMAP and node2vec sometimes generate unintuitive clusters, especially at lower weights, which are the most interesting ones for the following cluster analysis.

In Les Miserables, the tSNE approach achieves good separation of the clusters based on class label throughout all weight settings. UMAP, on the other hand, puts all nodes into two seemingly unintuitive clusters for lower weight settings. At $w = 50$, the clusters are separated better, but not as nicely as in tSNE. node2vec seems to have a similar problem at lower weight settings, where two to three clusters are identified correctly and the rest are joined into one.

Since Facebook is an unlabeled dataset, it is not as straightforward to check the validity of the clusters. No clusters are apparent for tSNE, which creates a "hairball"-like structure. UMAP and node2vec create similar layouts as before, where UMAP joins the nodes in two clusters and node2vec creates one big ball with some smaller clusters outside of it. For $w = 50$, the layout is already dominated by the force between disconnected pieces described a few paragraphs prior.

For Webk, tSNE and UMAP behave similarly with both achieving good separation based on class label on settings 30 and 50. node2vec suffers from the thresholding error described before at lower settings. At $w = 50$ though, it generates the layout with the best separation out of all the test results.

Looking at computational performance, tSNE and UMAP mostly finished in a similar timeframe during the experiments. Node2vec on the other hand took much longer to generate the embeddings, which is not surprising considering the much greater complexity of the method. tSNE and UMAP are also a lot easier to use in regard to parameter complexity.

Even though node2vec seemed like an ideal fit for the application at the start of this thesis, it falls behind simpler downprojection methods regarding speed and configuration complexity. This was especially true for the WebK dataset, which has 1702 attributes. The algorithm would need to create a virtual node for every one of these attributes, which would take an intolerable amount of time for an interactive application. This is why for that method, the number of attributes was reduced to only 50 using tSNE.

The final reason why node2vec is considered unfitting for this thesis, compared to other methods, is that it embeds both topological and attribute information since its random walks cover both real and virtual nodes. The idea behind the IGL-application was to strictly separate the two sources of information and then to produce a weighted combination of them. With node2vec, the user can not simply prioritize the embedding information, since it inherently includes the topological structure.

The two compound baselines, though sometimes performing well in the metrics, created visualizations that failed in effectively combining the two approaches and were not serving the goal of this thesis. The last applied method "overwrites" the transformations of the previous one. Even adapting the parameters, like different numbers of iterations for both steps, did not mitigate that effect.

9 Conclusion

The goal of this thesis was to provide an interactive application for the visualization of attributed graphs that lets the user adapt the layout of the nodes. A solution was implemented that extracts the connection and attribute information separately and lets the user assign weighting to them. Adapting that weight parameter allows for smooth transitioning from one layout method to the other.

The results show that the application is best suited for labeled graph datasets. The weighting can be adjusted such that the output shows the node clusters that fit the user's preferences. The experiments suggest that, for the task of class separation, the ideal configuration lies somewhere between $w = 30$ and $w = 50$. Further, that range could offer a good layout for disconnected graphs; something which is a problem with traditional force-directed methods.

The application is made up of a pipeline of multiple different components and algorithms. Over the course of its development it has become clear that parameterising each individual component and finding the optimal set of parameters is a task that is both complex but also very influential regarding performance.

An example of that would be the threshold parameter for the truncation operation described in section 4.6. A possible improvement of the IGL-algorithm could be to make that parameter adjustable to the user along with the weight.

Summary: Despite the mentioned criticisms and limitations, the IGL-algorithm has shown to be capable of giving an overview of a given graph's structure and inspecting it in regard to its connections and contained neighborhoods. The interactive element is dynamic, responsive and allows for quick testing of different layouts.

References

[1] *Anaconda Software Distribution*. Version Vers. 23.7.3. 2023. URL: https://docs.anaconda.com/.

[2] Daniel Archambault, Tamara Munzner, and David Auber. "GrouseFlocks: Steerable Exploration of Graph Hierarchy Space". In: *IEEE Transactions on Visualization and Computer Graphics* 14.4 (2008), pp. 900–913. DOI: 10.1109/TVCG.2008.34.

[3] Giuseppe Di Battista et al. "Algorithms for drawing graphs: an annotated bibliography". In: *Computational Geometry* 4.5 (1994), pp. 235–282. ISSN: 0925-7721. DOI: https://doi.org/10.1016/0925-7721(94)00014-X. URL: https://www.sciencedirect.com/science/article/pii/092577219400014X.

[4] Mike Bostock. *A Better Way to Code*. 2017. URL: https://medium.com/@mbostock/a-better-way-to-code-2b1d2876a3a0.

[5] Mark Craven et al. "Learning to Extract Symbolic Knowledge from the World Wide Web". In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI '98/IAAI '98. Madison, Wisconsin, USA: American Association for Artificial Intelligence, 1998, pp. 509–516. ISBN: 0262510987.

[6] Sara Di Bartolomeo et al. *Designing Computational Evaluations for Graph Layout Algorithms: the State of the Art*. Mar. 2023. DOI: 10.31219/osf.io/ms27r.

[7] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph drawing by force-directed placement". In: *Software: Practice and Experience* 21.11 (1991), pp. 1129–1164. DOI: https://doi.org/10.1002/spe.4380211102. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211102. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102.

[8] Aditya Grover and Jure Leskovec. *node2vec: Scalable Feature Learning for Networks*. 2016. arXiv: 1607.00653 [cs.SI].

[9] Aric Hagberg, Pieter Swart, and Daniel S Chult. *Exploring network structure, dynamics, and function using NetworkX*. Tech. rep. Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[10] David Harel and Yehuda Koren. "Graph Drawing by High-Dimensional Embedding". In: *Graph Drawing*. Ed. by Michael T. Goodrich and Stephen G. Kobourov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 207–219. ISBN: 978-3-540-36151-0.

[11] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[12] Geoffrey E Hinton and Sam Roweis. "Stochastic Neighbor Embedding". In: *Advances in Neural Information Processing Systems*. Ed. by S. Becker, S. Thrun, and K. Obermayer. Vol. 15. MIT Press, 2002. URL: https://proceedings.neurips.cc/paper_files/paper/2002/file/6150ccc6069bea6b5716254057a194ef-Paper.pdf.

[13] Takayuki Itoh and Karsten Klein. "Key-Node-Separated Graph Clustering and Layouts for Human Relationship Graph Visualization". In: *IEEE Computer Graphics and Applications* 35.6 (2015), pp. 30–40. DOI: 10.1109/MCG.2015.115.

[14] Mathieu Jacomy et al. "ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software". In: *PloS one* 9 (June 2014), e98679. DOI: 10.1371/journal.pone.0098679.

[15] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[16] Stephen G. Kobourov. *Spring Embedders and Force Directed Graph Drawing Algorithms*. 2012. arXiv: 1201.3011 [cs.CG].

[17] Jérôme Kunegis. "KONECT: The Koblenz Network Collection". In: *Proceedings of the 22nd International Conference on World Wide Web*. WWW '13 Companion. Rio de Janeiro, Brazil: Association for Computing Machinery, 2013, pp. 1343–1350. ISBN: 9781450320382. DOI: 10.1145/2487788.2488173. URL: https://doi.org/10.1145/2487788.2488173.

[18] Yao Yang Leow, Thomas Laurent, and Xavier Bresson. "GraphTSNE: A Visualization Technique for Graph-Structured Data". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[19] Jure Leskovec and Julian Mcauley. "Learning to Discover Social Circles in Ego Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/7a614fd06c325499f1680b9896beedeb-Paper.pdf.

[20] Laurens van der Maaten and Geoffrey Hinton. "Viualizing data using t-SNE". In: *Journal of Machine Learning Research* 9 (Nov. 2008), pp. 2579–2605.

[21] Leland McInnes, John Healy, and James Melville. *UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction*. 2020. arXiv: 1802.03426 [stat.ML].

[22] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[23] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[24] M. E. J. Newman and M. Girvan. "Finding and evaluating community structure in networks". In: *Physical Review E* 69.2 (2004). DOI: 10.1103/physreve.69.026113. URL: https://doi.org/10.1103%2Fphysreve.69.026113.

[25] *node2vec Software Distribution*. Version Vers. 0.4.5. 2023. URL: https://github.com/eliorc/node2vecp.

[26] Vatsal P. *Node2Vec Explained*. 2022. URL: https://towardsdatascience.com/node2vec-explained-db86a319e9ab.

[27] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.

[28] Leixian Shen et al. *Graph Exploration with Embedding-Guided Layouts*. 2023. arXiv: 2208.13699 [cs.DS].

[29] Lei Shi et al. "Hierarchical Focus+Context Heterogeneous Network Visualization". In: *2014 IEEE Pacific Visualization Symposium*. 2014, pp. 89–96. DOI: 10.1109/PacificVis.2014.44.

[30] Andre Suslik Spritzer and Carla Maria Dal Sasso Freitas. "Design and Evaluation of MagnetViz—A Graph Visualization Tool". In: *IEEE Transactions on Visualization and Computer Graphics* 18.5 (2012), pp. 822–835. DOI: 10.1109/TVCG.2011.106.

[31] *UMAP Software Distribution*. Version Vers. 0.5.3. 2018. URL: https://github.com/lmcinnes/umap.

[32] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[33] C. Vasudev. *Graph Theory with Applications*. New Age International (P), 2006. ISBN: 9788122417371. URL: https://books.google.at/books?id=Nb4iycwAR2IC.

[34] E.A.B.S.G. Williamson. *Lists, Decisions and Graphs*. S. Gill Williamson. URL: https://books.google.at/books?id=vaXv%5C_yhefG8C.