Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Beatrix Tugyi

ADAPTATION OF AN EXPLAINABILITY ALGORITHM TO GRAPH NEURAL NETWORKS

SUPERVISORS

Bálint Gyires-Tóth, PhD

Dániel Unyi

BUDAPEST, 2023

Contents

Kivonat 4

Abstract 5

1 Introduction 6

2 Theoretical Background 8

2.1 Graph Neural Networks 8

2.1.1 Task types on graphs 8

2.1.2 Graphs as input data 9

2.1.3 Architecture of the Graph Convolutional Networks 10

2.1.4 Graph Isomorphism Networks (GIN) 12

2.1.5 Graph Autoencoder 13

2.2 Explainability in Machine Learning 13

2.2.1 Explainability methods for Graph Neural Networks 15

2.2.2 The INVASE Explainability algorithm 19

3 Proposed Method 23

3.1 Architecture plan 23

3.1.1 Motivation of the choice 23

3.1.2 INVASE-GNN 23

3.2 Evaluation techniques 26

3.2.1 Visualizations 27

3.2.2 Accuracy 27

3.2.3 True Positive Rate and False Discovery Rate 28

3.2.4 ROC-AUC score 28

3.2.5 Fidelity 28

3.2.6 Sparsity 29

3.3 Baseline models 29

4 Implementation 30

4.1 Datasets 30

4.1.1 BA-Shapes 30

4.1.2 Tree-Cycle 31

4.1.3 Tree-Grid 31

4.2 Frameworks and languages 32

4.3 Implementation details 32

4.3.1 Dataset-specific decisions ... 32
4.3.2 INVASE with Pytorch ... 34
4.3.3 INVASE-GNN ... 34
4.3.4 Parameters of the algorithm ... 35
4.4 Special aspects of the method ... 36
5 Experimental results ... 37
5.1 Visualizations ... 37
5.2 Objective metrics' guidelines ... 37
5.3 BA-Shapes ... 38
5.3.1 Metrics ... 39
5.4 Tree-Cycles ... 40
5.4.1 Metrics ... 41
5.5 Tree-Grid ... 42
5.5.1 Metrics ... 42
5.6 Evaluation of the INVASE-GNN algorithm ... 43
6 Future work ... 44
7 Summary ... 45
References ... 46

Kivonat

Napjainkban a mesterséges intelligencia (MI) számos kutatási területen robbanásszerű ütemben terjed és fejlődik. Az MI számos iparágban forradalmasította az információfeldolgozást és a döntéshozatali folyamatokat. A mélytanulás (*deep learning*) gráfokban való alkalmazása új és egyre nagyobb fókuszt kapó irányzat. Gráf struktúrájú adatok közé tartoznak például a közösségi hálózatok, molekulák szerkezetei vagy hálózati rendszerek. Ezeknek az adatoknak a topológiai szerkezete már az adathalmazba bele van kódolva. A gráf neurális hálózatok (GNN-ek) hatékonyan alkalmazhatók rájuk, azonban kihívást jelent működésük megértése és hatékony használatuk.

A neurális hálózatokat gyakran „fekete doboz" jellegűnek nevezik, ami annyit jelent, hogy nem adnak visszajelzést arról, hogy a modell miért dönt úgy, ahogyan. Ahol a döntések kritikus jelentőséggel bírnak, ez különösen nagy hiányt jelent. Ide tartoznak az egészségügyi, pénzügyi vagy jogi területek, ahol azt várjuk, hogy a modell átláthatóan, megbízhatóan és torzítás mentesen működjön. Mindezek miatt jelenleg sokan a magyarázhatóságot tartják a mélytanulás egyik legfontosabb kutatási területének. Ezek a módszerek már megjelentek GNN-ekre is, de a létező megoldásoknak korlátai vannak, különösen összetett gráfokon alkalmazva. Kutatásom célja egy új magyarázhatósági algoritmus létrehozása GNN-hez. Kiindulásként egy összetett magyarázhatósági modellt használok, az INVASE-t, amely tabuláris adatokat képes elemezni. Ennek lényeges tulajdonsága, hogy minden egyes bemeneti példányra külön-külön meghatározza, hogy mely jellemzők játszottak kulcsszerepet a döntésben. Jól működik egyszerű tabuláris adatokhoz, viszont a gráfokra való alkalmazása további kihívást jelent. A gráfok tulajdonságai és adatszerkezete is nagyon eltér a tabuláris adatoktól. Az algoritmussal olyan magyarázatokat kívánok generálni, amelyek pontosabbak, mint a korábban készült algoritmusoké. Teljesítményüket mesterségesen generált adathalmazokon hasonlítom össze. A magyarázatokat a gráfok vizualizációjával mutatom be, amelyek kiemelik a fontosnak tartott éleket és csúcsokat. Ezeket a módszereket egy gráf struktúrájú adathalmazon használva, a műveletek megbízhatóbbá és ellenőrizhetőbbé válhatnak. A magyarázatok révén új ismeretek is felfedezhetünk azáltal, hogy a bemeneti és kimeneti adatok között eddig ismeretlen összefüggéseket tárunk fel.

Abstract

Nowadays, in many research fields, the use of Artificial Intelligence (AI) is spreading and developing at a rapid pace. The methods based on AI have revolutionized information processing and decision-making processes in many industries. The utilization of deep learning on graphs is a relatively new and increasingly focused area. In this case, the inputs are graph-structured data, such as social networks, structures of molecules, or network systems, just to name a few. This way the information about the shape of the graph and its topological structure is encoded in the dataset. Graph Neural Networks (GNNs) can be effectively applied to such data, although, it is a challenge to understand and interpret their working operations and to use them effectively.

Neural networks are often called "black-box" models, which means that they do not provide feedback about why the model makes the decisions it does. This can make it difficult to apply the models in industry, especially in areas where decisions have critical importance. These include healthcare, finance, or legal fields, where we expect the model to work transparently, trustworthy, and without biases. Thus, many people consider explainability one of the most important research areas of deep learning. Explainability methods have already appeared for GNNs, but the existing solutions have their limitations, especially when applied to complex graphs. My research aims to create a new explainability algorithm for GNNs. For a starting point, I use a complex explainability algorithm, termed INVASE, which expects tabular data as input. Its essential feature is that it does not explain all inputs in general, like many other methods, but for each input instance separately. It can specify which features played a key role in the decision. It works well for simple tabular data but adapting it to graphs is a completely new challenge. The structure and properties of graphs are very different from tabular data. With the proposed algorithm, I aim to generate explanations that are more accurate than the ones generated by previous methods. I compare their performance on artificially created graphs. I present the explanations as transparent visualizations of the graphs, which highlight the edges and nodes that are considered important by the algorithm. These methods can make the performed operations more reliable and controllable. By the explanations, we can also discover new information about the given area by revealing previously unknown relationships between the input data and the output.

1 Introduction

Artificial Intelligence (AI) covers a broad area, including Machine Learning (ML), a part of which is Deep Learning (DL). Deep learning uses complex structures called deep neural networks, which have many layers to process information. They usually work with a large amount of data and excel in tasks such as image and speech recognition, natural language processing, and text and image generation. The use of DL is becoming more and more widespread in today's world, thanks to the continuous development of models, software tools, hardware and the increasing amount of data. Research fields are also expanding in pace, with more and more challenging problems becoming available. Graph neural networks (GNNs) are nowadays an increasingly popular and promising trend, capable of making various predictions on graph structured data. GNNs have demonstrated unparalleled prowess in capturing complex relational patterns and dependencies between entities, paving the way for groundbreaking advancements and applications. For example, they contributed to the development of recommendation systems by handling large graph-structured data efficiently [1] and have shown promise in drug discovery by aiding in the analysis of molecular structures [2]. However, as with many other machine learning models, the impressive predictive performance of GNNs comes at the cost of transparency, leaving users and operators struggling with "black-box" models. Explainability algorithms were designed to solve this problem. The idea is to provide an easily understandable explanation for the predicted results generated by deep neural networks. In order to understand how the model made its decision, it is important to examine the underlying assumptions. This is not only beneficial to the further development of models, but also to their use in industry (particularly in critical areas).

In this work, my aim is to design and develop a novel explanability algorithm for GNNs. The proposed algorithm can be used to generate explanations for any graph neural network, overcoming the "black-box" disadvantage mentioned above. It is based on an explainability algorithm for tabular data, called INVASE. Translating these algorithms from conventional neural networks suitable for processing structured, tabular data to GNNs presents a substantial scientific challenge due to the intricate web of interactions within graph data.

In the first chapter, I present the theoretical background and the fundamentals of the used deep learning architectures and methods. I also describe the results of previous studies in the field.

Then, I present the proposed architecture, the steps of the method and the motivation behind the decisions. I show the implementation and experiment details of the method and different datasets. This is followed by a description of the results obtained according to different metrics. Finally, I will discuss the way forward and plans for further development of the method.

2 Theoretical Background

In this chapter, I present the theoretical background of the architectures and algorithms that I used in this research.

2.1 Graph Neural Networks

Graph neural networks (GNNs) form a relatively new class of deep learning models. They are designed for processing and analyzing structured data represented as graphs, by learning and propagating information across its nodes and edges. GNNs come in various architectures designed for different graph-related tasks [3]. Some common ones include:

- Graph Convolutional Networks (GCNs) for neighborhood aggregation [4],
- Graph Attention Networks (GATs) for weighted neighbor interactions [5],
- Graph Recurrent Networks (GRNs) for temporal graph analysis [6],
- Graph Isomorphism Networks (GINs) for graph structure learning [7] and
- Graph Autoencoders for compression and reconstruction [8].

These architectures cover a wide range of graph data and applications. I have used GCN, GIN and Graph Autoencoder in this project, so I will go into more details about them.

2.1.1 Task types on graphs

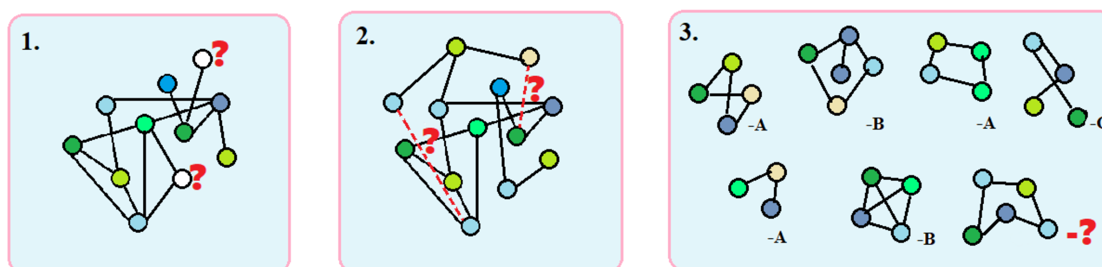
When working with graphs, we can distinguish three different types of tasks. These are summarised below and visualized in Figure 2.1. Further subdivisions and details can be found in [3].

1. Node prediction: For these tasks, the initial data is one graph, which may contain millions of nodes and/or edges. The aim is to predict some features of unknown or new nodes in this graph. For example, in the case of a social network, predicting the age and interests of a new unknown person when adding them to the social graph.

2. Edge prediction: As before, the input is one graph, but here the network's goal is to determine whether there is an edge between two selected nodes in the graph. For example,

in a social network, when two new people join, guessing whether there is a connection between them.

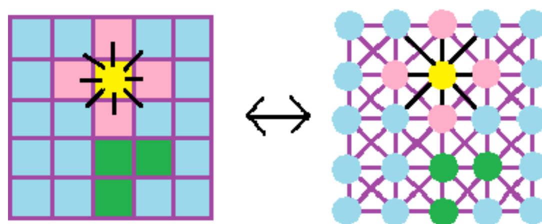
3. Full graph prediction: In this task, there are multiple graphs as inputs and the task is to predict the properties or values of the graph globally. For example, if our inputs are lots of small molecules, the task might be to estimate some property of a new molecule that we haven't seen before.



2.1 Figure - Illustration of the three types of tasks: node prediction (1, left), edge prediction (2, middle), and full graph prediction (3, right).

2.1.2 Graphs as input data

Graph structured data differs significantly from inputs commonly used in other machine learning models, such as image, sound, and text. It has the most similarities with images because graphs can be defined as generalisations of them. A two-dimensional image can be easily transformed into the input of a graph neural network, with pixels representing the nodes, and edges are drawn from each node to the adjacent pixels. This image to graph transformation is shown in Figure 2.2.



2.2 Figure - Comparison of image and graph inputs

In graphs, instead of nodes forming a regular shape, they have no fixed location and there is no limit to how many connections can start from a node or how many nodes and edges can be. These sets of data are defined only by the nodes and edges of the graph. Also, we cannot speak of their relative positions, since by convention there are no metrics for distances between the nodes of a graph. Additional properties and information can be

assigned to each node (node feature) and edge (edge feature). If necessary, we can encode locational information in them. A graph can be directed or undirected, depending on whether the links between the edges are unidirectional or bidirectional.

Graph structure can be a big advantage, because the topological structure of the data and the relations between nodes are already encoded in the dataset, which provides extra information. Many complex problems can be represented by graphs. There are a lot of graph-structured data available, for which these methods can be used. For example: network systems, social networks, recommender systems, molecules, or knowledge graphs. There are several ways to represent graphs by computers. For neural networks, the nodes of a graph are usually given as a matrix:

- X: A matrix with the same height as the number of nodes and width as the number of input features in the graph. Row x_i contains the feature vector of the i-th node.

There are two different approaches for the edges:

- Adjacency matrix (A): This is a square matrix of size equal to the number of nodes. Each element of *A[i, j]* can be either 1 or 0. 1 means that there is an edge between the i-th and j-th nodes, 0 means that there is not.

- **Edge index:** It is a two-dimensional array containing the indexes of the nodes of existing edges. Each edge is given by a pair of nodes' indexes, and these are placed in the array. So *edge_index[0, i]* contains one endpoint of edge i and *edge_index[1, i]* contains the other.

2.1.3 Architecture of the Graph Convolutional Networks

GCNs are a specialized subset of GNNs, bringing the power of convolutional operations to graph data by adapting concepts from image processing for graph-based tasks. The architecture of Graph Convolutional Networks that I used was presented by Thomas Kipf and Max Welling [4] in 2017. The main operation in this network is called Graph Convolution. Its inspiration is the Convolution used in the layers of a Convolution Neural Network (CNN) used for images [9]. How to map image input to graph data was introduced in the previous subsection. The interpretation of the convolution operation changes for GCNs. While in the case of images, this operation could be illustrated by a sliding window filter, here this analogy loses its meaning. The similarity is that Graph

convolution is also location-based: a given node only interacts with its adjacent nodes and aggregates the information obtained from them. The output of the network:

Z: a matrix of the same height as the number of nodes and width as the number of predicted features. Each element contains the predicted value corresponding to the given node and features.

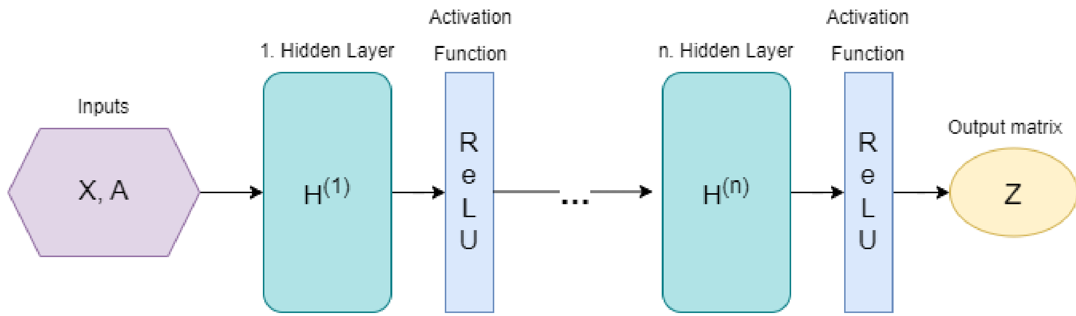
Between the input and the output of the network, there may be several hidden layers, which perform the graph convolution operation. Each hidden layer has its own weight matrix. We can obtain the next layer with the operation in (2.1).

$$H(l+1) = \sigma(A \cdot H(l) \cdot W(l)) \tag{2.1}$$

where W(l) is the weight matrix of layer l, H(l) is the l-th layers' output, H(0) is the input (X) and σ(.) is a nonlinear activation function, e.g. ReLU. In this step, the multiplication by matrix A aggregates all the properties obtained from all the neighbors. There is one problem with this: the value of the given node is omitted from the sum. This problem can be solved simply by adding an extra loop (self-loop) to each node, which can be achieved by adding the identity matrix to A ($\hat{A} = A + I$). This way, the node itself is included in the result. Another problem that arises is that matrix A is not normalized, so multiplying by it may distort the scaling of H(l) too much. For symmetric normalization, a matrix $\widehat{D}$ ($\widehat{D} = \sum_i \hat{A}_i$) can be used to multiply A and normalize the sum of all rows in A to 1. Considering this the final operation can be seen in (2.2).

$$H(l+1) = \sigma\left(\widehat{D}^{-\frac{1}{2}} \widehat{A} \widehat{D}^{-\frac{1}{2}} H^{(l)} W^{(l)} \right) \tag{2.2}$$

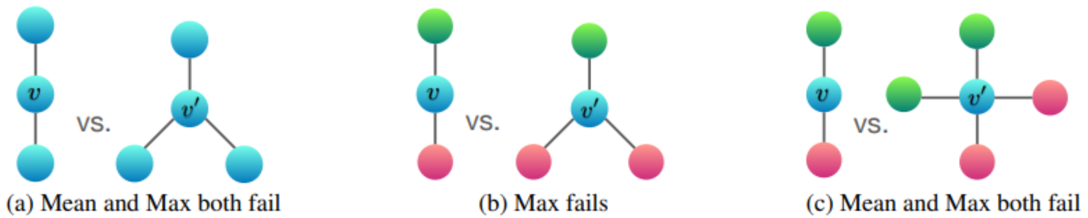
Any of these layers can be applied in sequence, taking into account that the size of an input of a layer is equal to the size of the output before it. There is no restriction on the number of hidden layers, but in practice only a few of them are used. The architecture of the graph convolutional network is visualized in Figure 2.3.



2.3 Figure – Architecture of a graph convolutional network

2.1.4 Graph Isomorphism Networks (GIN)

Graph Isomorphism Networks (GINs) [7] are specialized variants of GNNs. They are aiming to capture and learn from the intricate patterns and structures present in graph data. Unlike GCNs, GINs are designed to be more powerful in distinguishing different graph structures, addressing a limitation seen in many GNN models. While GCNs generally focus on aggregating neighborhood information to update the node's representation, they can struggle with distinguishish certain simple graph structures, potentially treating non-isomorphic graphs as if they were the same. Common aggregation functions include mean, sum, or max pooling. An example for this is shown in Figure 2.4.



2.4 Figure - Example of graph structures that other GNNs' (mean and max) aggregators fails to distinguish. Between two graphs, nodes v and v' get the same embedding, even though their different graph structure. Source: [7]

The core innovation of GINs lies in their unique an expressive approach to aggregating information, ensuring that subtle differences between graph structures are accurately captured. Specifically, the GIN aggregation function is defined as the (2.3) formula.

$$h_v^{(k)} = MLP^{(k)}\left(\left(1+\varepsilon^{(k)}\right)\cdot h_v^{(k-1)} + \sum_{u\in N(v)} h_u^{(k-1)}\right) \qquad (2.3)$$

where $h_v^{(k)}$ is the feature representation of node v at layer k, $N(v)$ is the set of neighbors of v, MLP is a Multilayer Perceptron neural network and ε is a learnable parameter, which adds more flexibility to the transform. This aggregation function enables GINs to excel in tasks requiring a deep understanding of graph topology. Through this, GINs open up new possibilities in graph analysis, providing a robust solution where the details of graph structure are of paramount importance.

2.1.5 Graph Autoencoder

One popular task of autoencoders is dimensionality reduction, but their utility extends beyond this, encompassing a variety of applications such as sparse autoencoders, denoising autoencoders, and more [10]. Even in cases where dimensionality reduction plays a significant role, the primary objective is often centered around learning a meaningful and efficient representation of the data.

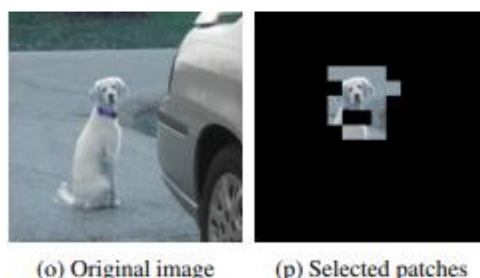
An autoencoder network's purpose is to learn the identity function: during the learning process, it compresses the data into a small latent space and then reconstructs the original input. This way, it can discover a more efficient and compact representation of the data. The first part of the network is called the Encoder, and the second is the Decoder. For graph data, I used a Graph Autoencoder [8], which is optimized specifically for graphs. The Encoder part usually consists of a GNN model. It takes the node features and the adjacency matrix of the graph as input and produces a latent vector representation for each node. The decoder aims to reconstruct the whole graph or some specified parts or properties. It could be a simple dot product between latent representations to predict if there is an edge between two nodes, or it could be a more complex GNN. This is a powerful tool for learning representations of graph-structured data, capturing both the local and global structure of a graph.

2.2 Explainability in Machine Learning

In the field of machine learning, the term explainability [11] refers to the ability of an AI model to provide understandable and interpretable explanations for its decisions and predictions. This is highly important for deep neural networks, because they usually operate as "black-boxes," making it challenging to understand the underlying mechanisms behind their decisions. This can make it difficult to apply such models in industry,

especially where they are used as part of human decision-making processes or in safety-critical applications. This includes health, financial, and legal areas where we expect the model to work transparently, reliably, and without bias. For example, in pharmaceutical research or a medical diagnosis, it's essential to understand why an AI system provides a specific diagnosis or recommendation. Similarly, if a model's task is to decide whether the convict is guilty or not, we may not know whether it is based on what they did, or perhaps discriminatorily, based on the ethnicity of the person. Therefore, in these applications, the existence of justifiability is crucial. In Europe, the AI Act [12] strongly regularises the use of AI in industry. There will be restrictions on existing companies using AI and requirements for explanations in many areas. Explainability algorithms increase the transparency and reliability of machine learning models and contribute to the principles of responsible artificial intelligence.

The growing number of better and better results creates a huge competition for the application of AI in the industry. How quickly and how well one can adapt and leverage new opportunities can be a huge advantage. However, it also carries a huge risk. Developers need to pay close attention to explainability so that the emerging models are as reliable as possible. With explainability, the focus is on finding a reasonable explanation for a network's decision, helping us to understand the steps involved. In the domains of images and text, researchers and developers employed various successful techniques to accomplish this, but in the domain of graphs, new methods have only recently begun to appear and are far from perfect.



2.5 Figure - An example result by an existing CNN Explanation algorithm. It is visible that it has successfully selected the parts necessary for the classification. Source: [13]

Explanation algorithms fall into two main categories:

1. Global-level explanation: These algorithms attempt to explain the entire functionality of the model. They help to understand what general rules, patterns, or trends

determine the decisions of the model. They do not take into account the diversity of the dataset.

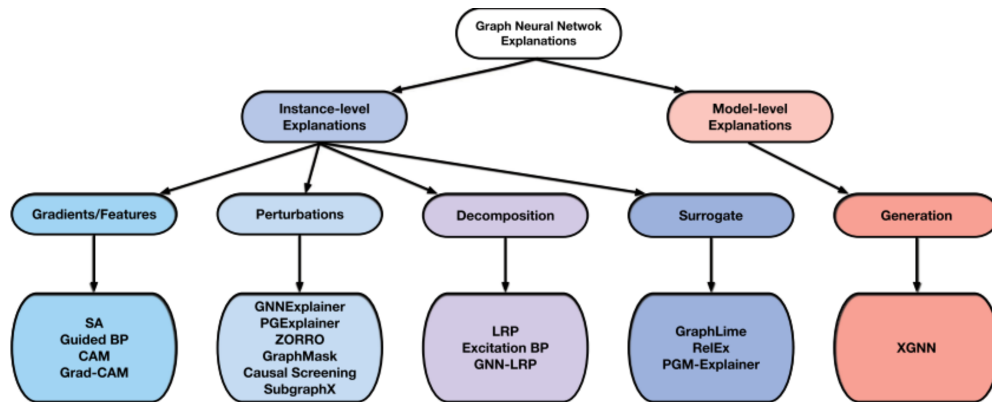
2. Instance-level explanation: These algorithms focus on explaining the individual predictions of the model. They help to understand why the model made a particular decision for a particular instance.

In addition to these, there are data-centric explanations, such as analyses conducted using techniques like UMAP and t-SNE [14]. They help in visualizing and interpreting the high-dimensional data the model is trained on, offering another layer of understanding and interpretability.

2.2.1 Explainability methods for Graph Neural Networks

The application of explainability with graph neural networks [17] is much more complex than the previous image and text-based networks' explanations. Those results are intuitive, and the understanding of the semantic meaning of input data is simple and straightforward. Humans can easily understand even highly abstract explanations. However, to understand graph-based models, domain knowledge of the datasets is necessary. Since graphs can represent complex data, it is challenging for humans to understand the meaning of them. Furthermore, in interdisciplinary fields like chemistry and biology, numerous mysteries remain unsolved, and there is still a deficiency in domain knowledge. For images and texts, we can study the important pixels or words. On the contrary, with graphs, we need to pay attention to important edges, nodes, and features of the nodes, thus the structural information and patterns of the graph can be much more significant than those.

Recently, several approaches have been proposed to explain the predictions of deep graph models. The two main categories from 2.2, for the general explainability technologies also appear here, there are instance-level and model-level explanations. The second category is still less explored. One of the reasons for that is that these methods give high-level explanations and only explain general behaviors, so they may not be human-intelligible since the obtained graph patterns may not even exist in the real world. In Figure 2.7 there is an overview of them.



2.7 Figure - An overview of the current graph explainability methods and their categorisation. Source: [17]

As shown in Figure 2.7, there are four more groups inside Instance-level explanations:

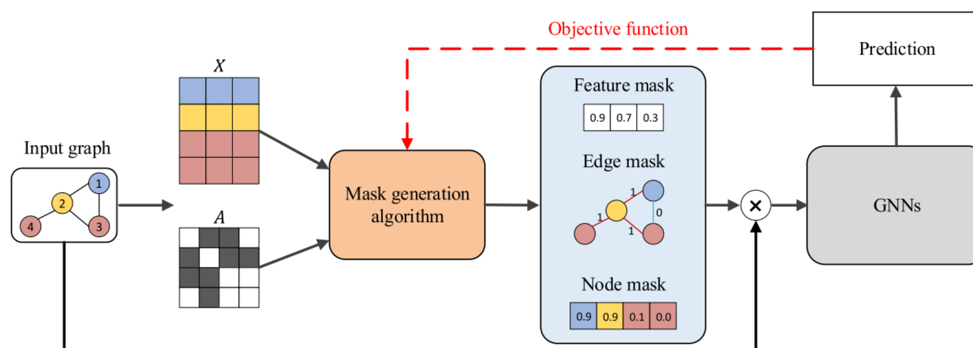
Gradients/Features-Based Methods

This technique is often used in image and text tasks [18]. The key idea is that, either to compute gradients of target predictions with respect to input features or to map hidden features to the input space through interpolation values to indicate to measure importance scores. Generally, larger gradients or feature values indicate higher importance. Both methods are closely tied to the model's parameters, so the explanations provided by them can accurately reflect the information embedded in the model. Since these methods are simple and general, they can be easily applied to graph-related tasks. There are several implementations of this type, including SA, Guided BP, CAM, and Grad-CAM. These methods all work based on the principle described above, the main difference between them is rooted in the process of gradient backpropagation and how distinct hidden feature maps are integrated.

Perturbation-Based Methods

These methods are widely employed to explain images. The key idea is, in order to study the important parts of the input, they monitor the change of the output prediction for various input perturbations. To create perturbations, different masks are obtained from the input graph to highlight crucial input features, including node masks, edge masks, and node feature masks, depending on the explanation task. After that, the generated masks are combined with the input graph. This new graph contains only the important input information fed into a trained GNN to evaluate the masks and update the mask generation process if needed. Intuitively, the important input features highlighted by the masks,

should carry essential semantic meaning and result in predictions close to the original ones. The main idea is visualized in Figure 2.8.



2.8 Figure - The key idea behind Perturbation-based explainability methods. Source: [17]

Many perturbation-based methods are proposed, including GNNExplainer [19], PGExplainer [20], ZORRO, GraphMask, Causal Screening, and SubgraphX. The main difference among these methods mainly lies in three aspects: the mask generation algorithm, the type of masks, and the objective function. The algorithm I have proposed also falls into this category.

Surrogate Methods

In these methods, the underlying idea is to fit a simple and interpretable surrogate model (for example a decision tree) to approximate the predictions of a complex deep neural network, but only for a sampled dataset from the neighbors of the given input example. These models work under the assumption that the relationships in the sampled neighborhood areas are less complex and can be well captured by a simpler surrogate model. Applying them to graph models is challenging since graph data is discrete and contain topology information. It is not trivial to choose how to determine the neighbouring regions and it is problematic to find a suitable surrogate model for them. Several surrogate methods are proposed, including GraphLime [21], RelEx, and PGM-Explainer. The key differences between these methods are how to obtain the local dataset and what interpretable surrogate model to use.

Decomposition Methods

These methods first decompose the model outputs' prediction scores (such as predicted probabilities), to the neurons in the last hidden layer. Then they backpropagate the scores layer by layer until the input space, while they usually ignore the activation functions. Then they use these decomposed scores as the importance scores. However, it

is challenging to directly apply such methods to the graphs since they contain nodes, edges, and node features. It is non-trivial to distribute scores to different edges while graph edges contain important structural information that cannot be ignored. Many decomposition methods have been proposed [22], including Layer-wise Relevance Propagation, Excitation BP and GNN-LRP. The main difference among these methods is the score decomposition rules, the view of decomposition, and the targets of explanations.

Model-level explanation

The only method in this category is called XGNN [23]. Instead of directly optimizing the input graph, its key idea is to explain GNNs via graph generation. It trains a graph generator so that the generated graphs can maximize a target graph prediction. This generated graph will be the explanation. We expect it to contain most of the important graph patterns. In XGNN, the graph generation is formulated as a reinforcement learning problem. For each step, the generator predicts how to add an edge to the current graph. Then the generated graphs are fed into the trained GNNs to obtain feedback to train the generator via policy gradient.

Table 2.1 shows a comprehensive analysis of the presented methods, based on six properties:

- *Type:* Indicates the main type of the explanation.
- *Learning:* Denotes whether a learning procedure was involved.
- *Task:* What tasks can the methods applied to. NC: Node classification, GC: Graph Classification.
- *Target*: The target of the explanations. N: Node, E: edge. NF: node feature, Walk: Graph walks.
- *Black-box:* Denotes whether the GNN treated as black-box during the explanation phrase.
- Flow: The direction of the computational flow of the explanations.

2.1 Table - A comprehensive analysis of different explainability methods. Source: [17]

Method	Type	Learning	Task	Target	Black-box	Flow
SA [54], [55]	Instance-level	✗	GC/NC	N/E/NF	✗	Backward
Guided BP [54]	Instance-level	✗	GC/NC	N/E/NF	✗	Backward
CAM [55]	Instance-level	✗	GC	N	✗	Backward
Grad-CAM [55]	Instance-level	✗	GC	N	✗	Backward
GNNExplainer [46]	Instance-level	✓	GC/NC	E/NF	✓	Forward
PGExplainer [47]	Instance-level	✓	GC/NC	E	✗	Forward
GraphMask [57]	Instance-level	✓	GC/NC	E	✗	Forward
ZORRO [56]	Instance-level	✗	GC/NC	N/NF	✓	Forward
Causal Screening [58]	Instance-level	✗	GC/NC	E	✓	Forward
SubgraphX [48]	Instance-level	✓	GC/NC	Subgraph	✓	Forward
LRP [54], [59]	Instance-level	✗	GC/NC	N	✗	Backward
Excitation BP [55]	Instance-level	✗	GC/NC	N	✗	Backward
GNN-LRP [60]	Instance-level	✗	GC/NC	Walk	✗	Backward
GraphLime [61]	Instance-level	✓	NC	NF	✓	Forward
RelEx [62]	Instance-level	✓	NC	N/E	✓	Forward
PGM-Explainer [63]	Instance-level	✓	GC/NC	N	✓	Forward
XGNN [45]	Model-level	✓	GC	Subgraph	✓	Forward

The results of the best performing of these different models are shown in the second last section compared to my algorithm's performance.

2.2.2 The INVASE Explainability algorithm

The INVASE (Instance-wise Variable Selection Using Neural Networks) explainability method [13] was developed in 2019. At that time global explanation models were widely used solutions, but this one implements instance-wise explanation. Therefore, this algorithm has a big advantage: it can flexibly discover key features for inputs of different sizes and characteristics. Unlike previous approaches, the number of elements chosen to be relevant can vary from sample to sample. It is important not to select too many features (over-explaining), as this will reduce the information of the obtained explanation. If a fixed number of best features were chosen globally, for some instances it would not be meaningful at all, it would simply perform well on the average of the samples. For example, different people have different relevant traits for a particular disease such as heart failure.Looking globally at a few traits these would not be discoverable.

The INVASE model consists of three different neural networks: the Selector, the Predictor, and the Baseline networks. It aims to learn which set of features are significant in a given prediction for each sample. By doing this, it can find the minimum information required for the network to make a correct decision. The resulting set of features is used to explain the decision, and it can also be useful to reduce the chance of overfitting. The INVASE algorithm is inspired by the Actor-Critic methods[15]. These are techniques used in the field of reinforcement learning, where two models, an Actor and a Critic,

collaborate in the learning process. The Actor is responsible for the execution of a set of rules, while the Critic evaluates its performance and gives feedback to the Actor to improve future decisions. In this scenario, the Selector network plays the role of the Actor and the Predictor the role of the Critic. While running INVASE, all three contained networks are iteratively trained using stochastic gradient descent.

The Baseline network is fundamentally a simple fully connected neural network that receives the original data as input and returns the predicted label as output. This network is used to reduce the variance, and it is also a common tool in reinforcement learning. A high variance would mean that the results of the training algorithm can vary greatly between trials, which can lead to instability and slow convergence. This model provides an estimation of the average reward that can be expected in a given state. The actual rewards obtained by the Actor-Critic method are thus corrected by the baseline value to reduce the variance of the expected rewards.

The Selector network is designed to minimize the KL (Kullback-Leibler) divergence[1] between the conditional probability distribution of the labels given all the features, and the conditional probability distribution of the labels given only the selected features (2.4). Intuitively, this makes the prediction with only the selected features as similar as possible to the prediction for the whole sample.

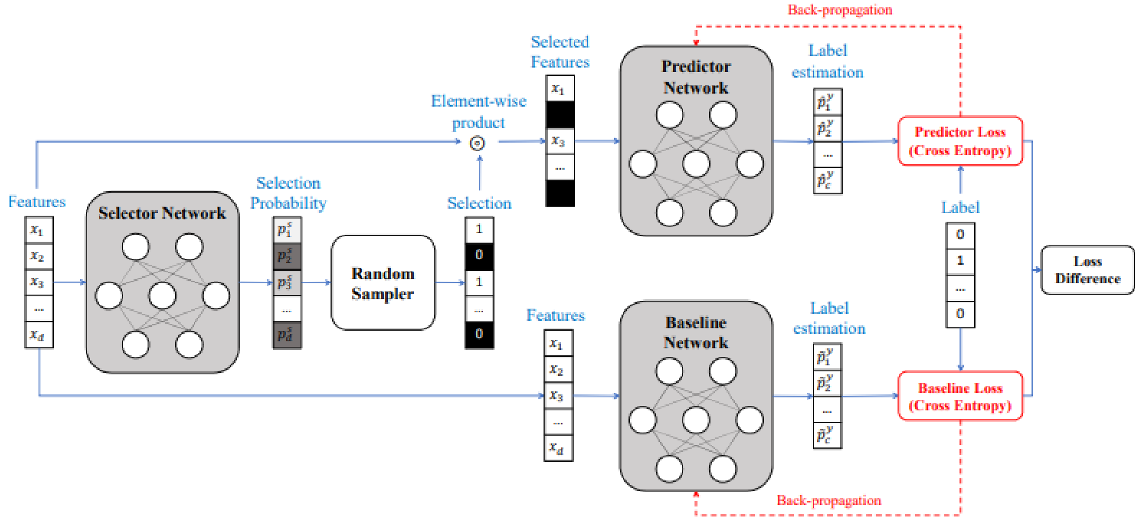
$$\min\{KL\big((Y|X)||(Y|X_{SELECTED})\big)\} \tag{2.4}$$

The input of the network is the original dataset, and the output is a mask of the same size as the input, consisting of 1s and 0s, indicating which ones are considered important. Its implementation consists of a fully connected neural network with a specialized loss function.

The architecture of the Predictor is identical to the Baseline's, the only difference is that its input is not only the data itself but also a mask produced by the Selector, which specifies which features should be considered in the current iteration. The first step of this network is to perform an element-by-element wise multiplication of this mask and the

[1] Kullback Leibler Divergence is a non-symmetrical measure of how a particular probability distribution P(x) varies from the baseline distribution q(x) over the similar values of discrete variable x.[16]

data. The obtained masked data is then passed through a fully connected neural network, like in the Baseline. Figure 2.6 shows the workflow of the detailed algorithm.



2.6 Figure - Architecture of the INVASE model. Source: [13]

The input data is first passed through the Selector network, which returns the importance value (selection probability) of each feature, from which a mask (selection) is generated by sampling and then multiplied with the original features. In this way, only the features selected by the Selector are retained from the data. The Predictor network takes the resulting vector as input and predicts the samples based on it. Similarly, the Baseline network makes a prediction, but it takes all the features into account without masking. Each network uses the difference between the received and original labels to backpropagate the error. Then we subtract the Baseline network's error from the Predictor network's error and update the Selector network with this result by backpropagation. Below is shown the pseudo-code of the algorithm, where the neural networks are denoted by f. θ, ϕ and γ are denoting the Selector, Predictor, and Baseline models, respectively

---

Pseudo-code of INVASE (Source: [13])

Inputs: learning rates (α, B), mini batch size (n_{mb}), dataset (D)

Inicialize parameters: $\theta\ (Selector), \phi\ (Predictor), \gamma\ (Baseline)$

While *Converge* **do:**

Sample one mini batch from the dataset: $\left(x_j, y_j\right)_{i=1}^{n_{mb}} \sim D$ (2.5)

for j= 1… n_{mb} do:

Calculate selection probabilities:

$$\left(p^j_{1,\dots} p^j_d\right) \leftarrow \hat{f}^{\theta}\left(y_j\right) \quad (2.6)$$

Sample selection vector:

for j= 1…d do: $s_i^j \sim Ber(p_i^j)$ (2.7)

Calculate loss:

$$(x_j, s_j) \Leftarrow \left[\sum_{i=1} y_i^j \, log\left(f_i^{\phi}\left(x_j^{(s_j)}, s_j\right)\right) - \sum_{i=1}^{c} y_i^j \, log\left(f_i^{\gamma}(x_j)\right)\right] \quad (2.8)$$

Update the selector network parameters (θ):

where $\lambda\|s_j\|$: L2 Regularization parameter and

$$\pi_\theta(x, s) = \prod_{i=1}^{d} \hat{f}_i^{\theta}(x)^{s_i} \cdot \left(1 - \hat{f}_i^{\theta}(x)\right)^{1-s_i} \quad (2.9)$$

$$\theta \leftarrow \theta - \alpha \frac{1}{n_{mb}} \sum_{j=1}^{n_{mb}} \left(\hat{l}_i(x_j, s_j) + \lambda\|s_j\|\right) \nabla_\theta \, log \, \pi_\theta(x_j, s_j) \quad (2.10)$$

Update the predictor network parameters (ϕ):

$$\phi \leftarrow \phi - \beta \cdot \frac{1}{n_{Mb}} \sum_{j=1}^{n_{mb}} \sum_{i=1}^{c} y_i^j \times \nabla_\phi \log\left(f_i^{\phi}\left(x_j^{(s_j)}, s_j\right)\right) \quad (2.11)$$

Update the baseline network parameters (γ):

$$\gamma \leftarrow \gamma - \beta \frac{1}{n_{mb}} \sum_{j=i}^{n_{mb}} \sum_{1=1}^{C} y_i^j \times \nabla_\gamma log\left(f_i^{\gamma}(x_j)\right) \quad (2.12)$$

Adapting this explainability algorithm to graph neural networks is challenging and demand specialized and flexible approaches due to the complex nature of graph data. Unlike tabular data, where features are independent, in graphs the connections between nodes are crucial. GNNs require an understanding of these connections to provide accurate explanations. For instance, in certain scenarios, an edge becomes significant solely when there's an alternative route present in the graph, creating a cycle. Only when these two elements are both considered can we precisely foresee the labels of the nodes. Therefore, their combined influence can't be accurately represented by just adding up their separate impacts. Furthermore, the varied sizes and structures of graphs add complexity, making it difficult to apply algorithms designed for fixed-size, grid-like tabular data directly to GNNs.

3 Proposed Method

In this chapter I present the architecture of a new graph explainability algorithm based on INVASE, which I termed INVASE-GNN. I go into details about the architectural and implementational decisions taken and the motivation behind them.

3.1 Architecture plan

I detailed the three main types of operations on graphs in the introductory section, and for each of them I would like to make available a separate version of the algorithm in the future. For this research, I only aimed for the implementation of node prediction. I developed the architecture for this, then optimized and tested it. The other two methods' architecture will be fundamentally the same, so the explainability method is not affected by which type I work with. They only need minor changes, which are entirely specific to the different graph tasks. For this reason, I focus on the algorithm itself, not the task type.

3.1.1 Motivation of the choice

INVASE is a remarkable method that works particularly well on tabular data. I have read previous research that has successfully applied the method to images [13]. The underlying idea of the algorithm itself, is not so much dependent on what input data is used. This gave me the idea that I could extend this to graph input data. No one had ever tried this approach before. I thought it was worth investigating what results could be obtained on graphs. Graph explainability is a very important research direction nowadays, where there are no perfect algorithms yet, so I also consider it an important research direction.

3.1.2 INVASE-GNN

The starting state for creating the INVASE-GNN, was the INVASE algorithm. This consists of three main parts: the Selector, Predictor, and Baseline models. I will go through them one by one, detailing the needed changes.

The simplest to design was the **Baseline**. This was transformed into a simple GNN model with the input being the nodes and edges of a graph, and the output being the

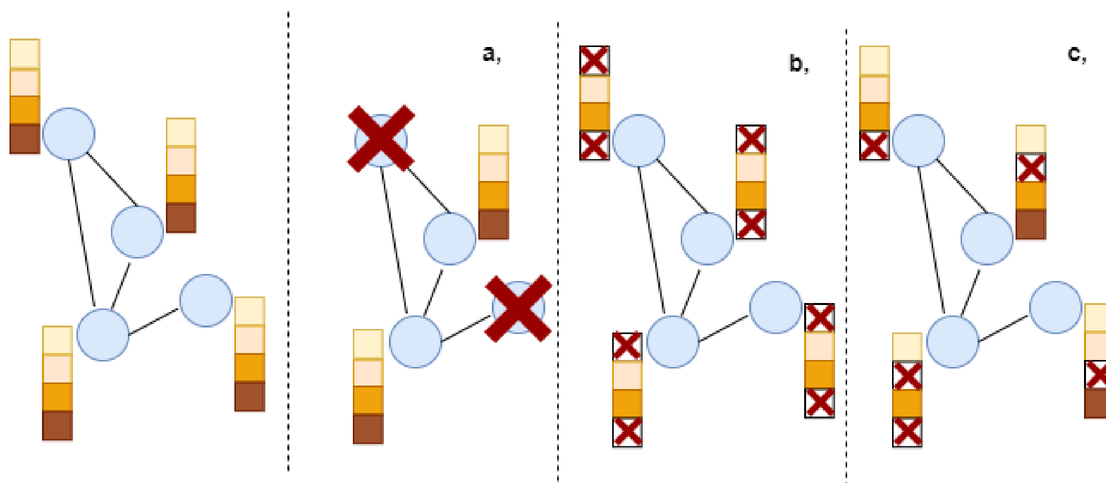
prediction of the nodes' labels. I replaced the fully connected layers of the model with GNN layers, the activation function between them remained ReLU.

The main part of the new **Predictor** network is structured the same way as the Baseline. The fully connected layers were also replaced by a graph neural network with a few layers. A significant difference here, is that the model not only receives as input the nodes and edges, but also a mask for each of them. These masks are selected by the Selector network. I multiply these masks with the original input element by element and train the network with only the remained part of the data. In practice, this is the part of the data that the model considers important in the current iteration. The output is also a prediction of the nodes' classes.

Designing the **Selector** network proved to be a more difficult task. In contrast to the original model, it is not only responsible for generating masks for the right features, but also for handling the edges and nodes. To implement this network, I used a graph autoencoder network. The encoder part receives the entire graph as input and outputs a lower dimensional representation of the same graph, also called a latent vector. The role of the decoder network is then to reconstruct the original graph from this latent vector as accuratly as possible.

The Decoder's output is two matrices, one for the nodes and its features and one for the edges, with the importance of each of them. From these, I can select the important elements using Bernoulli sampling, exactly as in the original article. Based on the number of selected features in each node I can also create a node mask, only keeping nodes with more important features than a certain threshold.

With nodes, the question arises of how to interpret their masks. There are several possible solutions. If nodes have more than one attribute, we have three options [24]. We can set the mask for all nodes, for the same attributes of each node, or we can let the network skip any attribute of any node without any predefined rule. These options are shown in Figure 3.1.

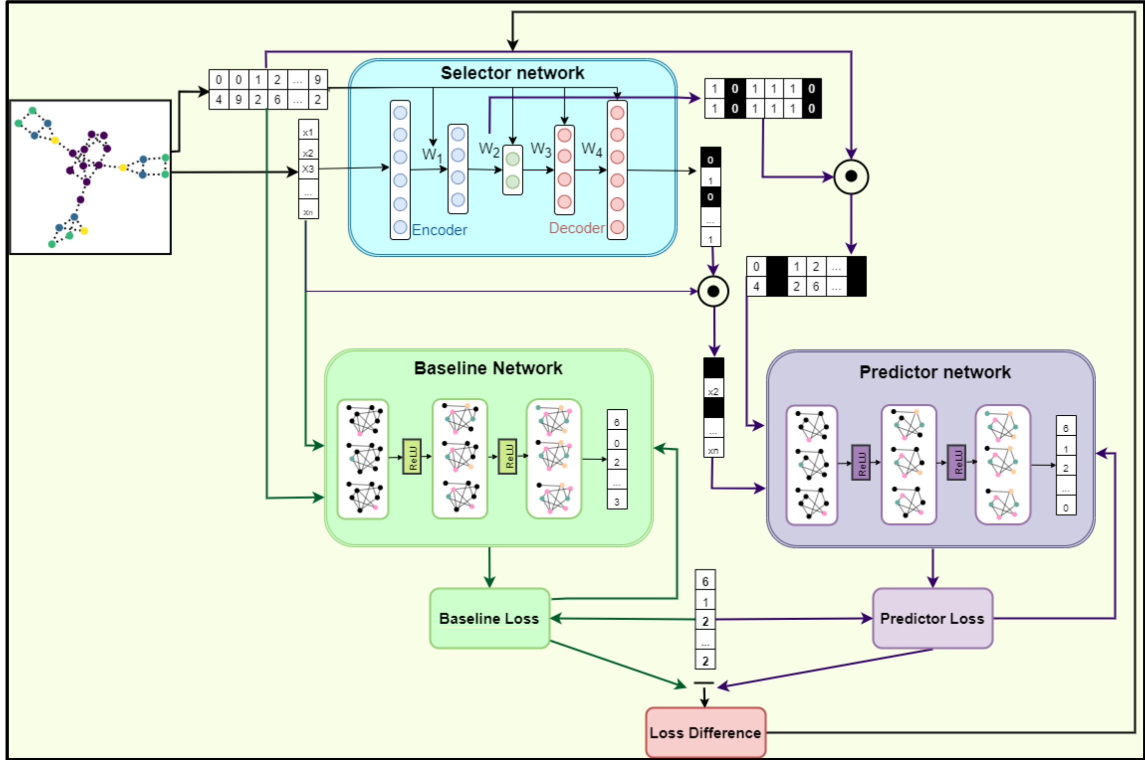


3.1 Figure – Possibilities for node masking. a: whole node masks, b: same feature for each node, c: any feature for each node.

An important issue with edges was how to handle whether or not there is an edge between two nodes. Not only the fact that two nodes are connected can be important for a decision, also the absent edges can be equally important. However, for this, the edge index list of real edges would not be enough. The full adjacency matrix would have to be used, which has the huge disadvantage of increased data volume, since in general networks have far fewer edges than non-edges. In addition to the increased size, the influence of the real edges would also be significantly reduced, which could degrade the performance of the algorithm. Therefore, I decided to focus only on edges for now, ignoring non-edges, but this may have to be modified later. The network needs to be trained for many iterations, during which time the Selector network will be able to learn how to identify the important edges and nodes.

The Selector's loss function also required a modification. I had to add to the original INVASE's loss the loss of the reconstructed edges and nodes from the autoencoder. I also had to include in the loss function that the network should attempt to select as few edges and nodes as possible that would still give an accurate prediction. This functionality was only included for the features in the original code. The magnitude of the selected features' size was multiplied by a hyperparameter lambda, and this was added to the loss. I used the same lambda hyperparameter and multiplied it with the number of non-zero elements of the created edge and node mask. Finally, I added this penalty to the loss value too.

During inference, instead of the Bernoulli sampling, only edges which probability are above a certain threshold (e.g. p=0.5) are kept. A visualization of the new architecture is shown in Figure 3.2.



3.2 Figure – The steps of the proposed INVASE-GNN algorithm

It is important to note that in the example when I mention graph neural network, it can be implemented by any type (e. g GCN, GAT, GIN) without any problem. The specific parts of the explanation algorithm are independent of the type of the used network.

3.2 Evaluation techniques

In this section I detail the evaluation metrics that are usually used for explainability in general and in graph explainability.

I use three different approaches to validate the results. First, I visualize the important and unimportant edges to make them illustrative and easy to understand. Secondly, I evaluate them according to the same aspects as in the original INVASE paper, like accuracy and ROC-AUC scores. I detail them in the next sections. These metrics cannot always be calculated for graph explanations since we need the ground truth

explainability scores or masks of the original dataset. For real datasets, these values usually unknown, but for synthetically generated datasets they may be available, based on the generation rules. Each of the synthetic datasets I used is composed of a basic graph with randomly attached motifs of a certain shape. This means that we can treat nodes and edges of the motifs as important and the others as unimportant. In general, the non-motif nodes labelled with 0. By deleting them, I can successfully extract the ground truth set of important edges. This way we can compare the predicted important edges and nodes with those in the ground truths. It's important to note that even if the algorithm doesn't find all of these, or finds some others, it can still perform well. Determining the true importance of the structure of a graph is more complicated than that, but it can give a good approximation.

Finally, to compare the results with other explainability graph neural network algorithms, I must consider the metrics that were used in those algorithms [17], such as Fidelity+, Fidelity- and Sparsity. These are the most relevant values for this graph explainability, so I mostly focussed on analysing and optimising these values.

I split the graph's nodes into test and training sets with a ratio of 80-20. I performed the training on the training dataset and the following evaluations on the test dataset.

3.2.1 Visualizations

One of the most intuitive ways of evaluating is to visualise the graph, using different colours to indicate edges and nodes that are considered important and unimportant. On the BA-Shapes dataset I would like to see the house motifs, on the Tree-cycle I would like to see the ring motifs and on the Tree-Grid I would like to see grid motifs marked with the same colour. Most of the elements outside of these should fall into another type. From these diagrams it is easy to see which parts of the graph are malfunctioning, and where the model could be improved.

3.2.2 Accuracy

This is the most common metric for classification. It means the ratio of the well-predicted elements and the number of the elements. For evaluating explanations, we can also compute the accuracy using only the important or unimportant parts of a graph.

3.2.3 True Positive Rate and False Discovery Rate

The True Positive Rate (TPR, Sensitivity) and False Discovery Rate (FDR, Specificity) are also frequently used to evaluate classifications. Their formulas are:

$$TPR = TP / (TP + FN), \quad FNR = FN / (TP + FN) \qquad (3.1)$$

where TP (True Positive) is the number of cases correctly identified as positive, FP (False Positive) is the number of cases incorrectly identified as positive and FN (False Negative) is the number of cases incorrectly identified as negative. For evaluating explanations, these metrics should be used in a different way, than in the regular classification tasks. Here we measure how well the algorithm found the important edges. The important edges are the positive and the unimportant edges are the negative cases. So, intuitively, TPR monitors how many percentages of all important edges are identified, and FDR monitors the percentages of those identified as important, that are not.

3.2.4 ROC-AUC score

The ROC (Receiver Operating Characteristic) curve is a graphical plot that illustrates the TPR and FDR values of a classification system as the threshold value is varied. The AUC (Area Under the Curve) represents the area under this curve. In a classification problem, the ideal value of it is 1 and the worst is 0,5.

3.2.5 Fidelity

To evaluate explanations it is important to use a metric that depends only on the model and the dataset, such as Fidelity+/-. Fidelity+ is defined as the difference of the original accuracy and the accuracy the model can achieve when we mask out the important features. Higher values indicate that the explanation of our model is better.

$$Fidelity +^{acc} = \frac{1}{N}\sum_{i=1}^{N}\left(\mathbb{I}(\hat{y}_i = y_i) - \mathbb{I}\left(\hat{y}_i^{1-m_i} = y_i\right)\right) \qquad (3.3)$$

where N is the number of graphs, $\mathbb{I}$ is the Indication function (its value is 1 if the argument is true, otherwise 0), y_i is the original label, $\hat{y}_i$ is the predicted label for the whole graph, $\hat{y}_i^{1-m_i}$ is the predicted labels used only the unimportant edges.

Fidelity- is defined as the difference of the original accuracy and the accuracy the model can achieve when we mask out the unimportant features. Lower values indicate that the explanation of our model is better.

$$Fidelity -^{acc} = \frac{1}{N} \sum_{i=1}^{N} \left(1(\hat{y}_i = y_i) - 1(\hat{y}_i^{m_i} = y_i) \right) \quad (3.4)$$

3.2.6 Sparsity

Good explanations should also be sparse. We want the capture them with the minimal number of edges and nodes. The Sparsity metric measures this property, by computing the fraction of the number of important elements (m_i) and the number of all elements M_i (3.7). We can compute this metrics for the edges, nodes, or node features separately.

$$s = \frac{1}{N} \sum_{1=1}^{N} \left(1 - \frac{|m_i|}{|M_i|} \right) \quad (3.7)$$

The Sparsity and Fidelity+/Fidelity- are highly correlated. Both are determined by a threshold value that tells you above what level of importance an item is considered important. Intuitively, with larger threshold values we can identify fewer items as important, so the Sparsity score can increase. On the other hand, fewer selected items probably decrease the Fidelity+ score. In order to fairly compare different explanation methods, it is recommended to use the Fidelity+ scores with the same threshold as the Sparsity scores.

3.3 Baseline models

To verify that our model has truly learned valuable importance information, it is helpful to compare it with a basic technique, a random model. Specifically, we assign the random importance scores to different edges and/or nodes and compute the performance of the new model to this random baseline. If our model performs better than the random one, it shows that it has truly understood important patterns in the data. This comparison not only validates our model but also helps pinpoint areas for improvement, ensuring its reliability and effectiveness in practical applications. In short, it's a quick sanity check that ensures our model is on the right track.

Furthermore, the Fidelity, Accuracy and Sparsity values are calculated for several existing GNN explainability methods, and it is worth comparing the results with those.

4 Implementation

In this chapter I describe the main step of the implementation of the designed architecture and the used datasets.

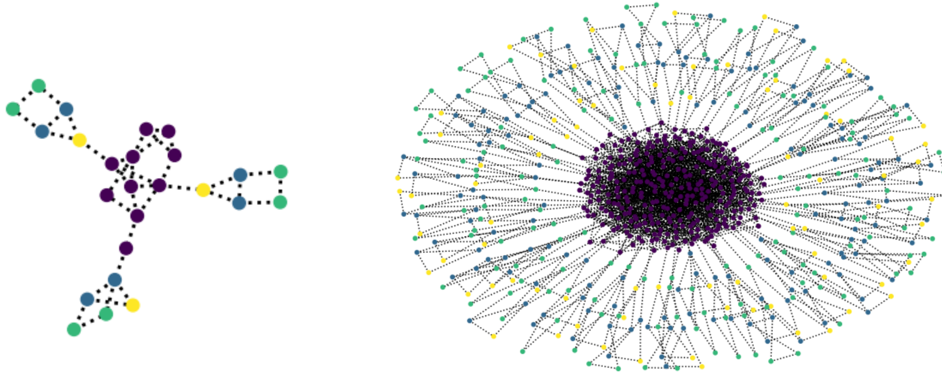
4.1 Datasets

In the dataset selection a key consideration was that, in order to evaluate the results, I needed a dataset that could be validated as an explanation, but not too easy for the network to obtain. The algorithm able to run on any dataset, even real graph datasets, but at the moment I could not measure the accuracy of the explanation on these. One use-case of the algorithm will be precisely to look for the importance values on a dataset where they are unknown. I have therefore chosen synthetic graph datasets to test the algorithm. I used the same datasets as many other explainability methods before[17][19], therefore my results will be comparable with them. I selected three different dataset, each consists of one graph. These graphs are random-generated based on some special rules.

4.1.1 BA-Shapes

This dataset contains one graph, which can be parameterized in terms of its size. The first step to create this graph is to construct a Barabási-Albert (BA) graph [25]. This is a random graph that is formed by making new nodes more likely to join well-connected nodes, a method called the Preferential attachment mechanism. This formation mechanism can be observed in real-world networks, such as social networks or the Internet, where popular nodes have more connections than less popular ones, following the power law in their degree distribution. As the next step, a stack of house-shaped motifs is generated, each consisting of five nodes. These motifs are then attached to random nodes in the graph. And finally, additional random edges are added to the graph to add some random noise. The labels for the nodes are divided into four classes: nodes with label 0 belong to the basic BA graph, nodes with labels 1, 2, 3 are placed separately in the middle, bottom or top of the houses. My visualization of it is shown in Figure 4.1.

As a parameter we can specify how many elements it should consist of. For this I used the same parameters used in the papers on other explainability methods [19], so the original BA graph consists of 300 nodes and 80 house-shaped patterns connected to it.



4.1 Figure - Visualization of the BA-Shapes dataset (Left: a small one with 3 house motifs, Right: The one I used for training with 80 house motifs)

4.1.2 Tree-Cycle

Tree-Cycle graphs consists of a base balanced tree graph and several six-node cycle motifs. These components are randomly connected. The label for the nodes in the tree graph is 0 otherwise 1. The depth of the tree and the number of motifs can be specified with parameters. I used the same parameters as other methods. The depth of the tree is 8 and the number of motifs is 60.

4.1.3 Tree-Grid

It is the similar to the Tree-Cycle dataset, except that it's employs nine-node grid motifs. The label for the nodes in the tree graph is 0 otherwise 1. I also used the same parameters as other methods. The depth of the tree is 8 and the number of motifs is 80.

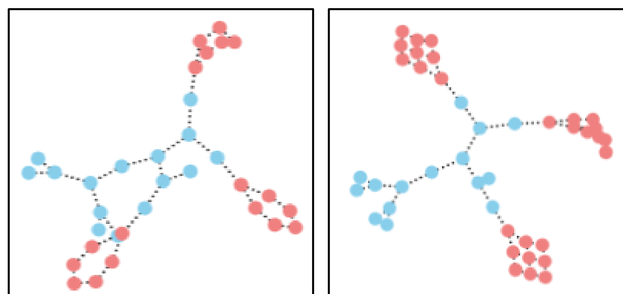


Figure 4.2 - Visualization if the Tree-Cycle (Left) and Tree-Grid (Right) datasets, with 3 motifs and a 3-depth tree.

4.2 Frameworks and languages

Python is the most popular language for deep learning models nowadays, and this is what I used too to write my code. There are two Python based libraries commonly used for deep learning on graphs:: Deep Graph Libary (DGL) and PyTorch Geometric (PyG). I chose the second library, because it has a module called Explainability, which was released in January 2023 [26] .This allows to handle such algorithms in a uniform and simple way, which is practical for me because it helps to easily implement and evaluate existing algorithms and compare them with my own. The PyTorch library has the advantage of being more flexible and customizable than other deep learning libraries. In addition, PyG is only compatible with this, so I decided to use it instead of the other well-known framework, Keras.

For the visualizations of the graphs, I used the NetworkX Python package [27]. This is for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It provides a standard programming interface that is suitable for many applications. The visualizations it produces can be personalized, they are understandable and easy to interpret.

4.3 Implementation details

The implementation of the code was performed according to the principles previously described, but some framework-, and dataset-specific modifications were necessary, which are discussed in this section.

4.3.1 Dataset-specific decisions

I had to adjust the designed algorithm to the selected datasets, which required two important changes.

4.3.1.1 Node features

These graph datasets do not contain any features in the node attributes, so the classification is entirely based on the structure of the graph. However, other graphs may contain node features as well, therefore I also prepared the proposed algorithm for them. The mask used for node features is largely identical to the one used in the original INVASE code. I implemented the node feature mask-specific parts but did not use them

for the presented experiments with the datasets. The most important aspect of using the algorithm on graphs is to be able to consider the importance of the edges and relationships of nodes. So, my goal was primarily to master this ability in my algorithm. The clarity of its extent can be seen when no node features are used at all. The presented datasets are exactly created for this purpose.

To test if the node feature mask generation process is working well, I added dummy features to the BA-Shapes dataset. With this new dataset, the algorithm successfully created node masks, which are shown in Figure 4.3. We can obtain the node mask based on this feature mask. To interpret the nodes' masks, I chose the method where I let the network skip any attribute of any node without any predefined rule.

	0	1	2	3	4
0	1	1	0	1	1
1	1	0	1	1	1
2	0	1	0	1	1
3	0	0	0	0	0
4	0	0	0	0	1
5	1	1	0	0	0
6	1	0	1	0	1
7	1	0	1	0	0
8	0	1	0	1	0
9	1	1	1	0	1
10	0	0	0	1	1
11	1	0	0	1	1

Figure 4.3 Created node mask for dummy node features. It shows five features' mask for the first 12 nodes.

4.3.1.2 Edge index symmetry

The graphs shown are all undirected. In Pytorch Geometry, they are interpreted by including all edges twice, in both directions (Source-To-Target, Target-To-Source) in the edge index variable. Thus, during aggregation, the message passing is performed in both directions. This way if I create the edge mask for the edges, I may obtain two different probability values for the same edge. However, this is not an expected behavior. Because of the undirectedness of the graph, I should not let the edge in one direction take on a different importance than the other.

In order to achieve this, I implemented the algorithm by creating a new variable for the datasets, in which each edge appears only once. With the decoder of the actor model, instead of the edge index, I generate the edge probabilities for this variable. Then I sample this result to obtain the edge masks. This way, the previously mentioned problem cannot occur.

The GNNs still expect the edge index as input, but I can easily apply this mask on that, in such a way that the same edges get the same mask. This way I achieved that the mask is symmetric for the different orientations of edges appearing in the edge index.

4.3.2 INVASE with Pytorch

The implementation of the original INVASE method was available only in Keras [28]. As a first step, I implemented the INVASE in Pytorch. Since the Keras and PyTorch libraries handle networks and their error backpropagation in a fundamentally different way, and the algorithm is sufficiently complex, the implementation was not trivial. It required a lot of consideration of how the different steps would work under the different frameworks. This implementation was essential, to make it suitable for GNNs, because to achieve that, it is necessary to look deep into the code. In such cases Pytorch is much more flexible, has many more customizable functions and the learning process is much more tractable. In addition, there are many more tools available to handle graphs and GNNs in Pytorch. In the overall result, the new code ran as well and with the same capabilities as the original, made it a suitable starting point for further work.

4.3.3 INVASE-GNN

The initial state for the novel INVASE-GNN code was the INVASE reimplementation in Pytorch, as described in the previous subsection. To perform the implementation, I followed the steps of the architecture plan, that I demonstrated in Chapter 3. Translating the theory into practice was challenging. Graph algorithms and models are inherently complex, so ensuring accurate implementation as per the pre-planned design was hard, due to potential mismatches between theoretical assumptions and real data and the used programming frameworks' capabilities. I had to determine how the ideas I had in mind, such as edge masking, could be implemented in code. A solid understanding of PyG was required to effectively implement and troubleshoot the algorithm. The correct implementation of the backpropagation algorithm was particularly difficult, since I trained three different networks simultaneously, with pieces of information being passed between them during the iterations, but the networks' errors had to propagate back independently. It required a deep understanding of how gradient backpropagation and automatic derivation operations work in Pytorch. To achieve the intended results, a lot of time had to be spent on optimization to finding the right

hyperparameters. Since no such model existed before, I had to experiment a lot until I got promising results and then it took a lot of finetuning to perfect it. I made sure to exclude all hyperparameters as input parameters to make the algorithm easy to optimize and customize. The completed algorithm returns an evaluation of the algorithm according to the previously mentioned metrics, as well as a visualization with the important and unimportant edges marked. The implemented code is available in a public Github repository.[2]

4.3.4 Parameters of the algorithm

The method has several hyperparameters, which effect its performance. By optimizing and tuning these, I was able to obtain the results detailed in the next chapter. I'll now go through the parameters one by one, detailing what they affect and the ranges within which they should be set for the used datasets.

Parameters of the training:

- **Dataset**: the name of the dataset, which define which dataset to use. Currently available values are “ba-shapes”, “tree-cycles” and “tree-grid”.
- **Data.x**: The features in each node. For the currently used dataset it is 1 for all nodes. I also experimented with adding the eigenvectors of the graph Laplacian as features, for example to examine the node features.
- **Iteration:** The number of iterations to train the networks. All three networks learn simultaneously in each iteration. Its value was between 5000 and 10000.

Parameters of the critic/baseline network:

- **Critic_model:** This is a string parameter, which define which network to use as the Critic and Baseline. Currently it can be “GCN” or “GIN”.
- **Critic_h_dim:** This parameter defines the size of the hidden dimensions used in the GNNs. In practice this was between 10 and 20.
- **Critic_n_layer:** This parameter defines the number of layers of the GNNs. I used 3 layers for a GCN model and 2 for a GIN model.

[2] https://github.com/TBeatrix/INVASE_GNN

- **Learning rate:** This parameter is responsible for the volume of the gradient update in the networks. It was a number between 0.1 and 0.0001.

Parameters of the actor network:

- **Actor_model:** This is a string parameter, which define which network to use in the actor as the encoder model. Currently it can be “GCN” and “GIN”.
- **Actor_n_layer** This parameter defines the number of layers of the Encoders’ GNN. I used 3 layers for a GCN model and 2 for a GIN model.
- **Actor_h_dim:** This parameter defines the size of the hidden dimensions used in the Encoder. In practice this was between 10 and 20 and smaller than the *critic_h_dim.*
- **Learning_rate:** This parameter is responsible for the volume of the gradient update in the networks. It was a number between 0.1 and 0.0001. I usually used the same learning rate as in the Critic and Baseline networks.
- **Lamda:** This is a special parameter, that is used in an extra penalty term in the Actors’ loss function. Its purpose is to control the ratio of the Actors' loss and the loss function from an extra loss, that aims to select as few edges as possible. This equilibrium is very important, and the method is sensitive to the proper adjustment of this parameter. If the value is too high, the algorithm simply skips all edges, and if it is too low, it selects all the edges. I have kept its value between 0.05 and 0.0001.

Others:

- **Important_edge_threshold:** This parameter controls how many edges are selected as important. The algorithm chooses the edges, that has a probability above this threshold.

4.4 Special aspects of the method

This method is unique, since instead of selecting the top k feature, it can determinate the number of important features for each graph. However, we can still control this behavior: if we want to select more features, we should lower the penalty for the number of edges and increase it if we want fewer edges.

The algorithm can be used for any graph neural network and data set, not just the ones I used in the implementation.

5 Experimental results

In this chapter I present the results obtained by the algorithm according to different metrics. I experimented with a variety of parameter options and models to maximize the results. The following tables show the results obtained with the best configurations.

5.1 Visualizations

As I discussed in section 3.2.1. evaluating visualisations is a difficult and subjective task. For larger graphs, unfortunately, the method is not even applicable, as the result is very dense and not completely visible.

Originally, I expected the model to pick the different motifs (house, cycle, nine-grid) as the important parts of the graph, but it was an unreasonable expectation. Since in these tasks we are predicting the class for all nodes of a graph, the model is not even expected to consider only those edges that are part of the motifs, since in the GNN layers the massage passing goes through the edges. The reason that we do not always see the motifs' edges as important can be explained by the fact that on these synthetic datasets, the GNN model can achieve equally good results by selecting other edges. Thus, the imperfect selection is not just a possible flaw in the explainability algorithm, since if the GNN model does not focus on learning these motifs, they are not the ones we want to get. Our goal is not to return the perfect ground true edge importance but to return what the model thinks is important, and unfortunately, there may be a big difference between the two. For these reasons, a much better approach to measuring the correctness of the algorithm is to use objective measures, which are described below the figures.

It can be seen in the visualizations in the next sections, that even though the motifs are not the only parts of the graph that are considered important, they still seem to play a major role in them.

5.2 Objective metrics' guidelines

In the numerical evaluation, Fidelity +/- was found to be the best metric to compare the different models. Therefore, I present these values for each dataset in the tables below the visualizations. The aim is to achieve as high Fidelity+ and as low

Fidelity- as possible. The results can be compared to the outcome when the edge masks are randomly selected. As it is detailed more in section 3.3, this is the best way to validate that our model has learned valuable and important information. For each dataset, I have displayed the results of four randomly masked models with sparsity values of 0.4, 0.5, 0.6 and 0.8. INVASE-GNN does not operate on a predefined importance percentage but displays the edges it finds most important, which number can be parameterized. For a proper comparison, I present four results with sparsity values close to the random ones. Public results for previous explainability algorithms, are the only available for the BA-Shapes dataset. To compare them with the INVASE-GNN, I have also included in the table these previous results, which were reported in [17]. I have even reproduced their Sparsity versus Fidelity+/- diagrams and added the INVASE_GNN algorithm to that. The parameters of my dataset were the same as the one they used, so they can be truly compared.

5.3 BA-Shapes

The visualization of the BA-Shapes dataset can be seen in Figure 5.1. The important edges are highlighted with blue and the unimportant with red. 40 percent of the edges in the picture belong to the important (blue) group. The colours of the nodes represent their classes.

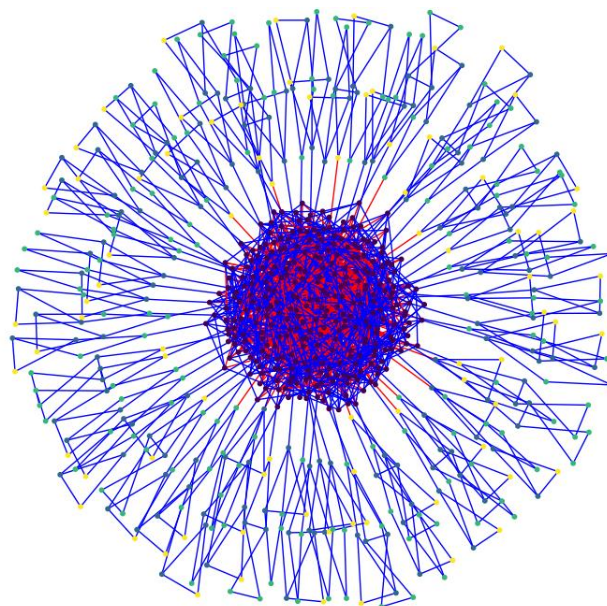


Figure 5.1 - Visualization of the whole BA-Shapes dataset with highlighted important (blue) and unimportant (red) edges

It can be seen that for this dataset, the model has learned perfectly well, and that the house motifs are important.

5.3.1 Metrics

5.1 Table – The results of the different methods on the BA-Shapes dataset

	Mask type	Sparsity	Accuracy of the whole graph	Accuracy on the important nodes	Accuracy on the unimportant	Fidelity+	Fidelity-
1	Random	0.4	0.8	0.62	0.55	0.25	0.18
2	Random	0.5	0.8	0.59	0.65	0.15	0.2
3	Random	0.6	0.8	0.45	0.63	0.17	0.35
4	Random	0.8	0.8	0.40	0.60	0.20	0.40
5	SubgraphX	0.6	N/A	N/A	N/A	0.3171	-0.0792
6	GNN-GI	0.6	0.8369	0.7051	0.6646	0.1723	0.1318
7	GNNExplainer	0.6	0.8786	0.8803	0.5861	0.2925	-0.0017
8	PGExplainer	0.6	0.7147	0.6489	0.5132	0.2015	0.0658
9	GNN-LRP	0.6	0.9243	0.9269	0.5857	0.3386	-0.0026
10	INVASE-GNN	0.43	0.81	0.6	0.47	0.33	0.21
11	INVASE-GNN	0.50	0.8	0.64	0.56	0.23	0.15
12	INVASE-GNN	**0.56**	**0.71**	**0.65**	**0.51**	**0.2000**	**0.0642**
14	INVASE-GNN	0.86	0.75	0.55	0.52	0.23	0.20

Presentation of the results in a form of diagrams can be seen in Figure 5.2.

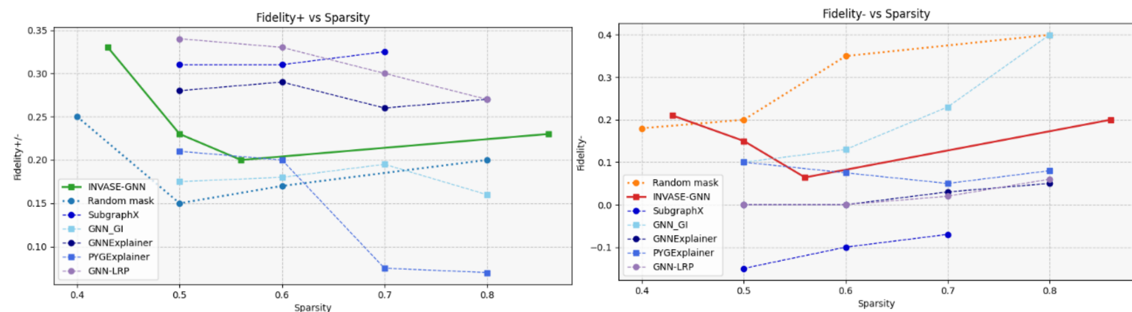


Figure 5.2 - Comparation of the Fidelity+ (left) and Fidelity- (right) scores of the proposed INVASE-GNN and the previous GNN explainability at different sparsity levels.

It can be seen from the table, or perhaps even more clearly from the attached diagrams, that the algorithm is proven to perform well on the BA-Shapes dataset. For all Sparsity values, our model achieved higher Fidelity+ and lower Fidelity- than the Random masking, which was the main goal. The diagrams shows that the achieved results are approaching the accuracy of previous algorithms, and even outperform some of them. The results are in the middle of the existing algorithms' performance range. This proves that the presented algorithm has a place among the existing ones and, with further improvement, may be able to show even better results than the others in the future.

5.4 Tree-Cycles

The visualization of the Tree-Cycles dataset can be seen in Figure 5.2. The important edges are highlighted with blue and the unimportant with red. 50 percent of the edges in the picture belong to the important (blue) group. The colours of the nodes represent their classes.

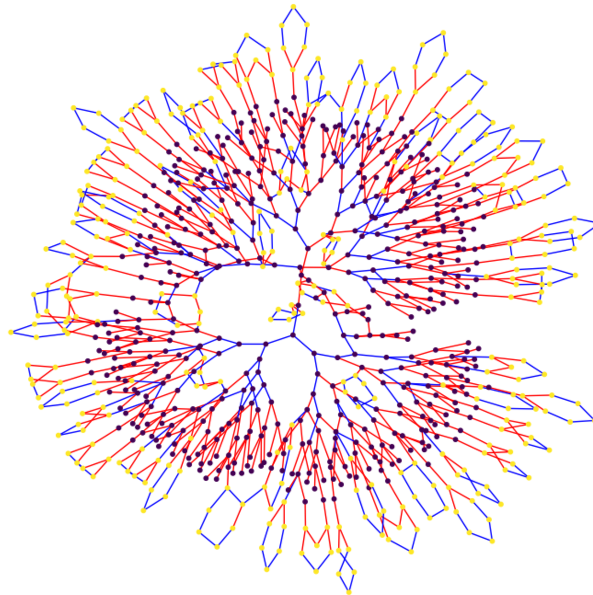


Figure 5.3 - Visualization of the whole Tree-Cycles dataset with highlighted important (blue) and unimportant (red) edges

It can be seen in the figure that for this dataset, the model does not fully learn the cycle motifs. Instead, it considers a few initial branches of the tree graph and the nodes of the cycle motifs that are not directly connected to the tree as important.

5.4.1 Metrics

5.2 Table - The results of the Random masked and the INVASE-GNN methods on the Tree-Cycles dataset

	Mask type	**Sparsity**	**Accuracy of the whole graph**	**Accuracy on the important nodes**	**Accuracy on the unimportant**	**Fidelity+**	**Fidelity-**
1.	Random	0.4	0.97	0.62	0.65	0.31	0.34
2.	Random	0.5	0.97	0.64	0.62	0.34	0.32
3.	Random	0.6	0.97	0.55	0.64	0.32	0.41
4.	Random	0.8	0.97	0.52	0.68	0.29	0.44
5.	INVASE-GNN	0.43	0.97	0.81	0.53	0.43	0.15
6	**INVASE-GNN**	**0.52**	**0.97**	**0.87**	**0.38**	**0.59**	**0.09**
7	INVASE-GNN	0.63	0.97	0.85	0.69	0.28	0.12
8	INVASE-GNN	0.74	0.98	0.68	0.63	0.35	0.30

The display of the table values on a line diagram:

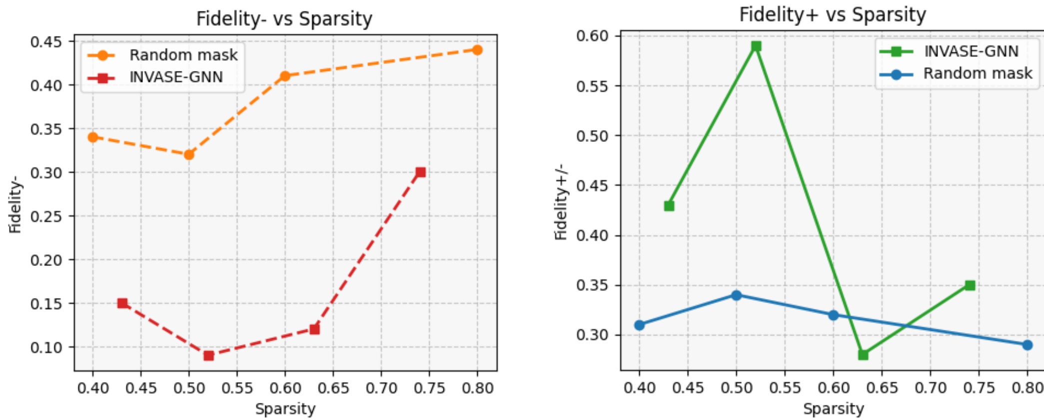


Figure 5.4 - The comparison of the Random mask and the INVASE-GNN mask in terms of Fidelity- (left) and Fidelity+ (right) with different sparsity values.

It can be seen from the table and the attached diagrams, that the algorithm is proven to perform well on the Tree-cycles dataset too. For all Sparsity values, our model achieved higher Fidelity+ and lower Fidelity- almost every time, which was the main goal. It is also quite intuitive to observe, that in row 6, 7 and 8 of the table, the edges chosen as important are less than the half of the edges, yet we achieve better prediction

on them. In the last one, only one quarter of the edges are used as important, and the obtained accuracy is still better than on the other three-quarter of the edges. This indicates that the algorithm has chosen very precisely what is important.

5.5 Tree-Grid

Unfortunately, due to the large size of the data set, it was not possible to see the whole graph in a transparent and understandable way. Instead, I selected three random point of the motifs and visualized their 4-hops subgraphs in Figure 5.2. A k-hops subgraph is a subgraph of nodes within k edges from a given point. The important edges are also coloured blue, and the unimportants are red.

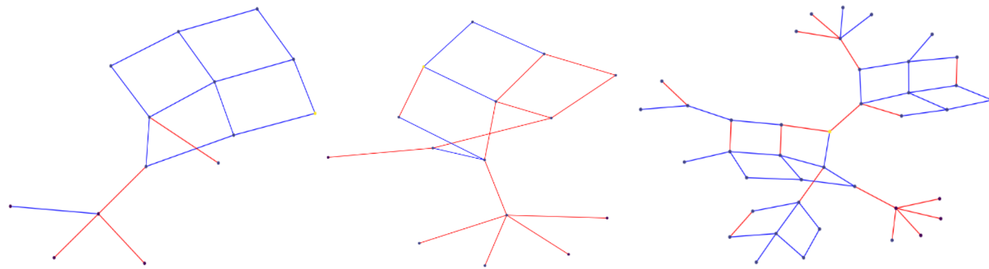


Figure 5.5 - Visualization of three random nodes (yellow nodes)' 4-hops subgraph of the Tree-Grid graph

It is shown in Figure 5.5 that the algorithm did not find all the edges of the motifs, but it did find a significant proportion of them.

5.5.1 Metrics

5.3 Table - The results of the Random masked and the INVASE-GNN methods on the Tree-Grid dataset

	Mask type	Sparsity	Accuracy of the whole graph	Accuracy on the important	Accuracy on the unimportan	Fidelity+	Fidelity-
1.	Random	0.4	0.817	0.62	0.61	0.19	0.17
2.	Random	0.5	0.817	0.57	0.61	0.18	0.23
3.	Random	0.6	0.817	0.6072	0.60	0.21	0.21
4.	Random	0.8	0.817	0.54	0.63	0.19	0.28
5.	INVASE-GNN	0.39	0.817	0.66	0.52	0.29	0.15

6	INVASE-GNN	0.5	0.81	0.68	0.53	0.28	0.13
7	**INVASE-GNN**	**0.67**	**0.821**	**0.68**	**0.54**	**0.27**	**0.13**
8	INVASE-GNN	0.79	0.821	0.55	0.60	0.21	0.27

The display of the table values on line diagrams:

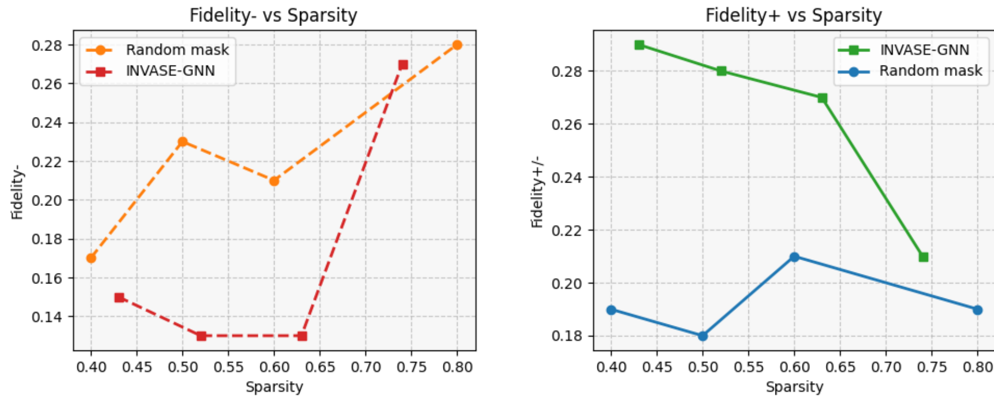


Figure 5.6 - The comparison of the Random mask and the INVASE-GNN mask in terms of Fidelity- (left) and Fidelity+ (right) with different sparsity values.

It can be seen from the table, or perhaps even more clearly from the attached diagrams, that the algorithm is proven to perform well on the Tree-Grid dataset. For all Sparsity values, our model achieved higher Fidelity+ and lower Fidelity-, which was the main goal. It is also quite intuitive to observe, that for example in row 7 of the table, the edges chosen as important are one third of the total, yet we achieve better prediction on them than on the remaining two thirds of edges.

5.6 Evaluation of the INVASE-GNN algorithm

The results have exceeded the expectations, they proved that the algorithm works well an each dataset. So, it can be claimed, that the proposed INVASE-GNN works successfully. It significantly outperforms the Random models and, as reflected by the BA-Shapes dataset, it is able to achieve results as good as previous algorithms. Like the previous approaches, it is still not able to achieve flawless results on graph neural networks. However, in the future, it could be a very useful baseline for achieving that.

The drawback is that the algorithm is very sensitive to the proper parameter settings and is not stable enough. But with careful attention and proper monitoring of the Fidelity values, the desired results can be obtained easily.

6 Future work

The results obtained are significant, which is evidence that the research direction is appropriate and well worth further investigation. I would like to extend the algorithm to datasets where not only the edges are available but also the nodes have different features. The difficulty is that there is currently no synthetic data set available to measure the goodness of this. As a next step, I would like to test the algorithm on real datasets, where the power and usefulness of the algorithm can be truly demonstrated. I would like to measure its performance on the other two important graph tasks: link prediction and graph prediction.

7 Summary

The popularity of graph neural networks is increasing with the rising growth of AI. However, neural networks are black-box models, and therefore their reliable and appropriate use requires efficient algorithms for explanation. There is an increasing focus on these explainability algorithms, and a direction called Explainable AI (XAI) is devoted to this. My goal was to derive such an explainability algorithm for graphs, based on a method called INVASE that works on tabular data. The adaptation of this model to the graph domain was far from trivial, with the challenges of the different structure and the special properties of graph data. The proper use of graph neural networks and their integration into the algorithm, as well as the choice of the appropriate architecture, was also a significant task. The designed steps and difficult points of the creation of the new method are detailed in Chapter 3. I implemented and tested the resulting architecture on three artificially generated datasets. To put this theory into practice, I had to make several additional decisions to adjust the method to the datasets, models, and used programming frameworks. This is presented in Chapter 4. The most interesting part is in Chapter 5, where I detail the achieved results based on several metrics. The results exceeded expectations and it can be clearly stated that the method works successfully. It achieves similar performance as previous graph explainability algorithms.

This result is of great significance, as the explainability of graphs is still a relatively new and significantly developing area. This solution, in addition to previous algorithms, may play a major role in further exploration of it. I would like to participate in this and contribute to the development of this field by further improving and extending the INVASE-GNN algorithm.

References

[1] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, Bin Cui, “Graph Neural Networks in Recommender Systems: A Survey”, arXiv:2011.02260

[2] Jiang, D., Wu, Z., Hsieh, CY. et al. Could graph neural networks learn better molecular representation for drug discovery? A comparison study of descriptor-based and graph-based models. J Cheminform 13, 12 (2021). https://doi.org/10.1186/s13321-020-00479-8

[3] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," AI Open, vol. 1, pp. 57-81, 2020, doi: 10.1016/j.aiopen.2021.01.001

[4] Thomas Kipf, „Graph Convolutional Networks”, https://tkipf.github.io/graph-convolutional-networks/ (access date: 2023 Oktober)

[5] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio, „Graph Attention Networks”, arXiv:1710.10903

[6] Seo, Y., Defferrard, M., Vandergheynst, P., Bresson, X. (2018). Structured Sequence Modeling with Graph Convolutional Recurrent Networks. In: Cheng, L., Leung, A., Ozawa, S. (eds) Neural Information Processing. ICONIP 2018. Lecture Notes in Computer Science, vol 11301. Springer, Cham. https://doi.org/10.1007/978-3-030-04167-0_33

[7] Keyulu Xu, Weihua Hu, Jure Leskovec, Stefanie Jegelka, ‘How Powerful are Graph Neural Networks?’, arXiv:1810.00826

[8] Thomas N. Kipf, Max Welling, “Variational Graph Auto-Encoders”, arXiv:1611.07308

[9] Y. LeCun, Y. Bengio, and G. Hinton, "An Introduction to Convolutional Neural Networks," arXiv:1511.08458, 2015. (access date: 2023 Oktober)

[10] Lilian Weng, “From Autoencoder to Beta-VAE”, https://lilianweng.github.io/posts/2018-08-12-vae/, (access date: 2023 Oktober)

[11] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you? Explaining the predictions of any classifier," arXiv:1602.04938v3, 2016.

[12] Think Tank, European Parliament, Artificial intelligence act, https://www.europarl.europa.eu/thinktank/en/document/EPRS_BRI(2021)698792 (access date: 2023 Oktober)

[13] J. Yoon, J. Jordon, M. van der Schaar, "INVASE: Instance-wise Variable Selection using Neural Networks," in Proceedings of the International Conference on Learning Representations (ICLR), 2019., https://openreview.net/forum?id=BJg_roAcK7 (access date: 2023 Oktober)

[14] Henry Han, Wentian Li, Jiacun Wang, Guimin Qin, Xianya Qin, “Enhance explainability of manifold learning”, https://doi.org/10.1016/j.neucom.2022.05.119.

[15] V. Konda and J. Tsitsiklis, "Actor-Critic Algorithms," in Proceedings of the Advances in Neural Information Processing Systems (NIPS), S. Solla, T. Leen, and K. M. Müller, Eds., vol. 12, MIT Press, 1999., https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf (access date: 2023 Oktober)

[16] ScienceDirect, Kullback-Leibler Divergence, https://www.sciencedirect.com/topics/engineering/kullback-leibler-divergence (2023 Oktober)

[17] H. Yuan, H. Yu, S. Gui and S. Ji, "Explainability in Graph Neural Networks: A Taxonomic Survey," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 45, no. 5, pp. 5782-5799, 1 May 2023, doi: 10.1109/TPAMI.2022.3204236. (access date: 2023 Oktober)

[18] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, “Grad-cam: Visual explanations from deep networks via gradient-based localization,” in 2017 IEEE International Conference on Computer Vision (ICCV). IEEE, 2017, pp. 618–626.

[19] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: generating explanations for graph neural networks. Proceedings of the 33rd International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA, Article 829, 9244–9255.

[20] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, Xiang Zhang, “Parameterized Explainer for Graph Neural Network”, arXiv:2011.04573

[21] Qiang Huang, Makoto Yamada, Yuan Tian, Dinesh Singh, Dawei Yin, Yi Chang, “GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks”, arXiv:2001.06216

[22] H. Yuan, L. Cai, X. Hu, J. Wang, and S. Ji, “Interpreting image classifiers by generating discrete masks,” IEEE Transactions on Pattern Analysis and Machine Intelligence, 2020.

[23] Yuan, Hao & Tang, Jiliang & Hu, Xia & Ji, Shuiwang. (2020). XGNN: Towards Model-Level Explanations of Graph Neural Networks. 430-438. 10.1145/3394486.3403085.

[24] Blaž Stojanovič, „Graph Machine Learning Explainability with PyG”, Medium cikk, https://medium.com/@pytorch_geometric/graph-machine-learning-explainability-with-pyg-ff13cffc23c2 (access date: 2023 Oktober)

[25] Bertotti, M.L., Modanese, G. The configuration model for Barabasi-Albert networks. Appl Netw Sci 4, 32 (2019). https://doi.org/10.1007/s41109-019-0152-1

[26] Pytorch Geometric, GNN Explainability, 2023, https://pytorch-geometric.readthedocs.io/en/latest/tutorial/explain.html, (access date: 2023 Oktober)

[27] Aric A. Hagberg, Daniel A. Schult and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX", in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008 (access date: 2023 Oktober)

[28] Jinsung Yoon, James Jordon, Mihaela van der Schaar, „INVASE implementation" https://github.com/jsyoon0823/INVASE (access date: 2023 Oktober)