Providing Bipartite GNN Explanations with PGExplainer

Tristan Marten Lewin Schulz

Bachelor of Science

May 19, 2025

Supervisors:
Prof. Dr. Stefan Harmeling
Lukas Schneider

Artificial Intelligence (VIII)
Department of Computer Science
TU Dortmund University

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Contents

Chapter 1

Introduction

1.1 Motivation and Background

MOTIVATION SAT! Advancements in deep learning SAT solving, e.g. NeuroSAT. Need for evaluation of these models, use of GNN explainers as SAT can be reduced to graph domain. Application of PGExplainer on NeruoSAT to generate explanations. Explanations need gt to be evaluated on accuracy. Use concepts like unsat cores and backbones as gt and compare to explanations provided by PGExplainer to see whether NeuroSAT explanations align with "human-observable" principles. Therefore explanation and replication of PGExplainer. After successful replication, application on NeuroSAT. Graph task: Prediction of UNSAT and unsat cores as gt. Node task: Difficult with NeuroSAT?

1.2 Thesis structure?

1.3 Related Work

TODO: MOVE TO BEFORE CONCLUSION?! MOVE TO THEORETICAL BACKGROUND?!
Original PGExplainer? We seek to provide a reimplementation using PyG framework and reproduce the results. TODO: Move criticism to discussion? Existing reimplementation of PGExplainer using PyG by Holdijk et al. Unable to achieve results of original implementation. Improvement on some datasets, considerable deterioration on BA-2Motif. Criticize approach of original and lacklustre documentation as well as differences between codebase and paper. We try own implementation to further improve results. We follow the configs used in the reimplementation but conduct sweeps ourselves around these and the original configs. Compare results to both implementations.
Taxonomy paper evaluates PGExplainer on Fidelty and achieves low scores, sides with reproducibility paper above: "is not performing as promising as its original reported results".

Propose only using PGExplainer for Node classification task.NeuroSAT as downstream

task for SAT experiments with PGExplainer. PyG reimplementation code provided by Rodhi. We create required data with provided methods, add unsat cores as gt and adapt the NeuroSAT model to allow passing edge weights into the adjacency matrix. PGExplainer adapted for SAT data, NeuroSAT embeddings and evaluation.

Chapter 2

Background

In this chapter we define the necessary background for understanding PGExplainer as well as the follow-up work regarding its application on the Boolean Satisfiability Problem (SAT).

2.1 Deep learning

In this chapter we introduce Deep Learning (DL) in the context of Machine Learning (ML) and their concepts required for this work. The definitions in this chapter loosely follow Goodfellow et al.[10].

An ML algorithm generally learns to perform a certain task from data. Common ML tasks include classification, where the goal is to assign an input to one of k categories, and regression, where the program shall predict a numeric value given some input.

ML algorithms can be broadly divided into supervised and unsupervised learning. In this work, we focus on supervised learning, meaning that the algorithm learns from a dataset containing both features and labels or targets that the algorithm is supposed to predict. In other words, the algorithm learns from a training set of inputs x and outputs y to associate another unseen input with some output.

DL entails expanding the size of the model used in our algorithm to allow for representations of functions with increasing complexity. This enables many human-solvable tasks that consist of mapping an input vector to an output vector to be performed with DL, given sufficiently large models and datasets.

In many cases DL involves optimizing a function, usually by minimizing $f(x)$. This objective function is also referred to as loss function in the context of minimization. To minimize a function $f(x)$ we make use of the derivative $f'(x)$ that tells us how to change x in order to get an improvement in y. We can reduce $f(x)$ by moving x in the direction opposite of the sign of its derivative, since $f(x - \epsilon \text{ sign}(f'(x))) < f(x)$ for small enough ϵ. This technique is called gradient descent (see Cauchy[3]).

Furthermore, a DL algorithm typically consists of the following components: a dataset specification, an objective function, an optimization procedure, like gradient descent, and a model.

2.1.1 Multilayer Perceptron

A classical DL model is the multilayer perceptron (MLP) with the general goal of approximating a function f^*. In the case of classification we could define a function $y = f^*(x)$ that maps an input x to a label y. The MLP then defines the mapping $y = f(x;\Theta)$ and learns the value of the parameters Θ that best approximate the function. These models are also referred to as feedforward neural networks, as they process information from x, through the intermediate computations that define f, to the output y without feedback connections that would feed outputs back to itself. The name network is derived from their representation as a composition of multiple different functions, that are described by a directed acyclic graph. An example network is $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ that consists of input layer $f^{(1)}$, hidden layer $f^{(2)}$ and output layer $f^{(3)}$. The length of this chain of functions is called depth and origin of the term ”deep learning”. The approximation is achieved by training our network with training data, that consists of approximated examples of $f^*(x)$ at different points in the training and labels $y \approx f^*(x)$. These training examples dictate the output layer to generate a value close to y for each x. The learning algorithm then learns to utilize the other hidden layers, without specified behaviors, to achieve the best approximation. It is to note that the hidden layers are vector-valued with each vector element, referred to as unit, loosely taking the role of a neuron in neuroscience. Models are therefore also referred to as neural networks.
A linear layer for our model with parameters Θ consisting of weight w and bias b can be described as $f(x; w, b) = x^T w + b$ for an input vector x. To keep the model from strictly learning a linear function of its inputs we can calculate the values of a layer h by applying an activation function g to its output to describe the features. This results in our hidden layer $h = g(W^T x + c)$, with W containing the weights of a linear transformation and c the biases.
The activation function usually serves the purpose of mapping to a real number between 0 and 1, imitating the activation of a neuron. We try to use activation functions that are continuous differentiable and easily calculated, to minimize computational complexity (see GNN book [13]). Examples are the Sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

and the Rectified Linear Unit (ReLU)

$$ReLU(x) = \begin{cases} 0 & x \leq 0, \\ 1 & x > 0. \end{cases} \tag{2.2}$$

2.1.2 Backpropagation

The back propagation algorithm is commonly used during neural network training. It optimizes the network parameters by leveraging gradient descent. It first calculates the values for each unit in the network given the input and a set of parameters in a forward order. Then the error for each variable to be optimized is calculated, and the parameters are updated according to their corresponding partial derivative backwards. These two steps will repeat until reaching the optimization target.
TODO: Concrete formulas with chain rule?

2.1.3 Computational graph

2.1.4 Regularization

2.1.5 Batchnorm, LayerNorm, ...

2.1.6 Monte Carlo Sampling

In order to best approximate a randomized algorithm we can make use of Monte Carlo methods as described in Goodfellow et al.[10][p.590]. A common practice in ML is to draw samples from a probability distribution and using these to form a Monte Carlo estimate of some quantity. This can be used to train a model that can then sample from a probability distribution itself.
More specifically, the idea of Monte Carlo sampling is to view a sum as if it was an expectation under some distribution and to approximate this estimate with a corresponding average. Let

$$s = \sum_x p(x)f(x) = E_p[f(x)] \tag{2.3}$$

be the sum to estimate with p being a probability distribution over a random variable x. Then s can be approximated by drawing n samples from p and constructing the empirical average

$$\hat{s}_n = \frac{1}{n}\sum_{i=1}^{n} f(x^{(i)}). \tag{2.4}$$

2.2 Graph Theory

TODO: edge weights, k-hop/computational graph?

These definitions will loosely follow Liu et al.[13]. A graph is a data structure consisting of a set of nodes that are connected via edges, modeling objects and their relationships. It can be represented as $G = (V, E)$ with $V = \{v_1, v_2 ... v_n\}$ being the set of n nodes, and $E \in V \times V$ the set of edges. An edge $e = (u, v)$ connects nodes u and v, making them neighbours. Edges are either directed or undirected and lead to directed or undirected graphs if exclusively present. The degree of a node v is the number of edges connected to v and denoted by $d(v)$. G can be described by an adjacency matrix $A \in \mathbb{R}^{n \times n}$, where

$$A_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

If G is an undirected Graph the adjacency matrix will be symmetrical.
Alternatively an undirected graph $G = (V, E)$ with n nodes and m edges can be represented as an incidence matrix $M \in \mathbb{R}^{n \times m}$, where

$$M_{ij} = \begin{cases} 1 & \text{if } \exists k \text{ s.t. } e_j = \{v_j, v_k\}, \\ 0 & \text{otherwise.} \end{cases}$$

We adopt the conventions from Diestel[7][p.2] to refer to the node and edges set of any graph G with $V(G)$ and $E(G)$ respectively, regardless of the actual names of the sets, as well as a referring to G with node set V as G on V. G is called a subgraph of another graph $G' = (V', E')$ if $V(G) \subseteq V(G')$ and $E(G) \subseteq E(G')$. This is denoted as $H \subseteq G$. The number of nodes in a graph $|V|$ is its order and the number of edges $|E|$ is its size. We additionally define bipartite graphs according to Asratian et al.[1]: A graph G is bipartite if the set of nodes V can be partitioned into two sets V_1 and V_2 so that no two nodes from the same set are adjacent. The sets V_1 and V_2 are called colour classes and (V_1, V_2) is a bipartition of G. This means that if a graph is bipartite all nodes in V can be coloured by at most two colours so that no two adjacent nodes share the same colour.

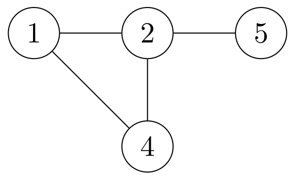

Figure 2.1: A simple undirected graph G with $V = \{1, ..., 5\}$ and $E = \{\{1,2\}, \{2,4\}, \{1,4\}, \{2,5\}\}$.

2.2.1 Random Graphs

Gilbert[9] describes the process of generating a random graph of order N by assigning a common probability to each potential edge between two nodes for existing in the graph. Note that these random selections are made independently of each other and effectively drawn from a Bernoulli distribution.
Version 1 (probability space for PGE not needed, definition for one random graph suffices?):
A random graph is further described by Diestel[7][p.323] as follows. Let $V = \{0, ..., n-1\}$

be a fixed set of n elements. Our goal is to define the set $\mathcal{G}$ of all graphs on V as a probability space, which allows us to ask whether a Graph $G \in \mathcal{G}$ has a certain property. To generate our random graph we then decide from some random experiment whether e shall be an edge of G for each potential $e \in V \times V$. The probability of success - accepting e as edge in G - is defined as $p \in [0, 1]$ for each experiment. This leads to the probability of G being a particular graph G_0 on V with e.g. m edges being equal to $p^m q^{\binom{n}{2}-m}$ with $q := 1 - p$.

Version 2:
A random graph is further described by Diestel[7][p.323] as follows. Let $V = \{0, ..., n-1\}$ be a fixed set of n elements. To generate our random graph we then decide from some random experiment whether e shall be an edge of G for each potential $e \in V \times V$. The probability of success - accepting e as edge in G - is defined as $p \in [0, 1]$ for each experiment. This leads to the probability of G being a particular graph G_0 on V with e.g. m edges being equal to $p^m q^{\binom{n}{2}-m}$ with $q := 1-p$. It follows our desired probability space $\mathcal{G} = (n, p)$ as the product space

$$\Omega := \prod_{e \in [V]^2} \Omega_e \tag{2.5}$$

with $\Omega_e := \{0_e, 1_e\}$, $\mathbb{P}_e(\{1_e\}) := p$ and $\mathbb{P}_e(\{0_e\}) := q$. TODO: This is probably unnecessary for PGE.

$$E(G) = \{e | \omega(e) = 1_e\} \tag{2.6}$$

E(G) = Edges of G. G is called a random graph on V with egde probability p.

2.3 Information Theory

To fully understand the learning objective of PGExplainer it is necessary to define the concepts of entropy and mutual information. We follow the definitions by Cover et al.[5][p.13] if not stated otherwise.

2.3.1 Entropy

Entropy is used to describe the uncertainty of a random variable. It measures the amount of information required on average to describe a random variable. Let X be a discrete random variable with alphabet $\mathcal{X}$ and probability mass function $p(x) = Pr\{X = x\}$ for $x \in X$. The entropy $H(X)$, also written as $H(p)$, is defined as

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x). \tag{2.7}$$

The log is to the base e and entropy is measured in nats in our case. TODO: DEFINE EDGE CASES log 0

The conditional entropy of Y given X is defined as the expected value of the entropies of the conditional distributions, averaged over the conditioning random variable. If $(X, Y) \sim p(x, y)$ for a pair of discrete random variables (X, Y) with joint distribution $p(x, y)$, the conditional entropy is defined as

$$H(Y|X) = -\sum_{x \in \mathcal{X}} p(x) H(Y|X = x) \tag{2.8}$$

$$= -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(y|x) \tag{2.9}$$

$$= -E \log p(Y|X) \text{ with E = Expectation.} \tag{2.10}$$

2.3.2 Relative Entropy and Cross-Entropy

The relative entropy between two distributions is a measure of "distance" between the two. It measures the inefficiency of assuming a distribution to be q when the true distribution is p. It is not a true measure of distance as it is among other things not symmetrical. The relative entropy takes a value of 0 only if $p = q$. We define the KL divergence or relative entropy between two probability mass functions $p(x), q(x)$ as

$$D_{KL}(p||q) = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \tag{2.11}$$

Suppose we know the true distribution p of our random variable. We could then construct a code with an average description length of $H(p)$. If we used the code for the distribution q instead, we would need $H(p) + D_{KL}(p||q)$ nats to describe the random variable on average. This is also referred to as the cross-entropy (see Goodfellow et al.[10][p.74]):

$$H(p, q) = H(p) + D_{KL}(p||q) \tag{2.12}$$

We derive for the discrete case with mass probability functions p, q defined on the same support $\mathcal{X}$:

$$H(p, q) = H(p) + D_{KL}(p||q) = -\sum_{x \in \mathcal{X}} p(x) \log p(x) + \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} \tag{2.13}$$

$$= -\sum_{x \in \mathcal{X}} p(x) \log p(x) + \sum_{x \in \mathcal{X}} p(x) \log p(x) - \sum_{x \in \mathcal{X}} p(x) \log q(x) \tag{2.14}$$

$$= -\sum_{x \in \mathcal{X}} p(x) \log q(x) \tag{2.15}$$

2.3.3 Mutual Information

Another closely related concept is mutual information. It measures the amount of information that one random variable contains about another or the reduction in uncertainty of said variable due to knowing the other. Let X and Y be two random variables with the joint probability mass function $p(x, y)$ and marginal probability mass functions $p(x)$ and $p(y)$. Mutual information $I(X; Y)$ is the relative entropy between the joint distribution and the product distribution $p(x)p(y)$:

$$I(X;Y) = \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{2.16}$$

$$= H(X) - H(X|Y) \tag{2.17}$$

2.4 Graph Neural Networks

Graph Neural Networks(GNNs)[17] are a DL-based approach that operates on graphs. Due to their unique non-Euclidean property, they find usage in classification, link prediction, and clustering tasks. Their high interpretability and strong performance have led to GNNs becoming a commonly employed method in graph analysis. They combine the key features of convolutional neural networks[12], such as local connection, shared weights, and multi-layer usage, with the concept of graph embeddings[2] to leverage the power of feature extraction and representation as low-dimensional vectors for graphs (see Liu et al.[13]).

Graphs are a common way of representing data in many different fields, including ML. ML applications on graphs can mostly be divided into graph-focused tasks and node-focused tasks. For graph-focused applications our model does not consider specific singular nodes, but rather implements a classifier on complete graphs. In node-focused applications however the model is dependent on specific nodes, leading to classification tasks that rely on the properties of each node. The supervised GNN model by Scarselli et a.[17] tries to preserve the important, structural information of graphs by encoding their topological relationships among nodes.
A node is naturally defined by its features as well as its related notes in the graph. The goal of a GNN is to learn state embeddings $\mathbf{h}_v \in \mathbb{R}^S$ for each node v, that map the neighborhood of a node into a representation. These embeddings are used to obtain outputs $\mathbf{o}_v$, that e.g. may contain the distribution of a predicted node label. The GNN model proposed by Scarselli et al.[17] uses undirected homogeneous graphs with $\mathbf{x}_v$ describing the features of each node and x_e the optional features of each edge. $co[v]$ and $ne[v]$ denote the set of edges and neighbors of node v respectively. The model updates the node states according to the input neighborhood with a local transition function f that is shared by all nodes. Additionally, the local output function g is used to produce the output of each node. $\mathbf{h}_v$ and $\mathbf{o}_v$ are therefore defined as

$$\mathbf{h}_v = f(\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]}), \tag{2.18}$$

$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v), \tag{2.19}$$

with $\mathbf{x}$ denoting input features and $\mathbf{h}$ the hidden state. $\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]}$ denote the features of the node v and of its edges, as well as the states and features of its neighboring nodes, respectively. We define $\mathbf{H}, \mathbf{O}, \mathbf{X}$ and $\mathbf{X}_N$ as the matrices that are constructed by stacking all states, outputs, features, and node features, respectively. This allows us to define with the global transition function F and the global output function G, which are stacked versions of their local equivalent for all nodes in a graph:

$$\mathbf{H} = F(\mathbf{H}, \mathbf{X}), \tag{2.20}$$

$$\mathbf{O} = G(\mathbf{H}, \mathbf{X}_N). \tag{2.21}$$

Note that F is assumed to be a contraction map and the value of $\mathbf{H}$ is the fixed point of equation (2.20). To compute the state the iterative scheme

$$\mathbf{H}^{t+1} = F(\mathbf{H}^t, \mathbf{X}) \tag{2.22}$$

is used with $\mathbf{H}^t$ denoting iteration t of $\mathbf{H}$. The computations of f and g can be understood as the feedforward neural network.
To learn the parameters of this GNN, with target information t_v for a specific node v, the loss is defined as

$$loss = \sum_{i=1}^{p} (t_i - \mathbf{o},) \tag{2.23}$$

where p are the supervised nodes. A gradient-descent strategy is utilized in the learning algorithm, which consist of the following three steps: the states h_v^t are updated iteratively using equation (2.18) until time step T. We then obtain an approximate fixed point solution of equation (2.20): $\mathbf{H}(T) \approx \mathbf{H}$. For the next step the gradients of the weights W are calculated from the loss. Finally, the weights W are updated according to the computed gradient. This allows us to train a model for specific supervised or semi-supervised tasks and get hidden states of nodes in a graph.

TODO: include figure of graph with neighborhood?

2.4.1 Convolutional Graph Neural Networks

Explain? Used in architecture of downstream task, only slightly relevant
GCN, GraphSAGE?

2.5 Perturbation-based Explainability in GNNs

Methods in DL have seen growth in performance in many tasks of artificial intelligence, including GNNs. However, the interpretability of these models is often limited due to their black-box design. Explainability methods aim to bypass this limitation by designing post-hoc techniques that provide insights into the decision-making process in the form of explanations. Such human-intelligible explanations are crucial for deploying models in real-world applications, especially when applied in interdisciplinary fields. There exist several different approaches for explaining predictions of deep graph models, that can be categorized into instance-level methods and model-level methods (see Yuan et al. [22]). Instance-level methods aim to explain each input-graph by identifying important input features for its prediction, leading to input-dependent explanations. These can further be grouped by their importance score calculation into gradients/feature-based, perturbation-based, decomposition methods and surrogate methods. Model-level methods, on the other hand, aim to explain GNNS without considering specific inputs, leading to input-independent, high-level explanations.

In this work we focus on the perturbation-based approach, more specifically the PGExplainer[14], that aims to evaluate the change of prediction with respect to input perturbations. The intuition behind this is that when input information crucial to the prediction is kept, the new prediction should roughly align with the prediction from the original input. The general pipeline for different perturbation based approaches can be described as follows: First, the important features from the input graph are converted into a mask by our generation algorithm, depending on the explanation task at hand. These masks are applied to the input graph to highlight said features. Lastly, the masked graph is fed into the trained GNN to evaluate the mask and update the mask generation algorithm according to the similarity of the predictions.

It is important to distinguish between soft masks, discrete masks and approximated discrete masks. Soft masks take continuous values between $[0, 1]$ which enables the graph algorithm to be updated via backpropagation. A downside of soft masks is that they suffer from the "introduced evidence" problem (see Dabkowski et al.[6]). Any mask value that is non-zero or non-one may add new semantic meaning or noise to the input graph, since graph edges are by nature discrete. Discrete masks however always rely on non-differentiable operations, e.g. sampling. Thus, the approximated discrete masks utilize reparameterization tricks to avoid the "introduced evidence" problem while also enabling back-propagation.

Explanations can on the one hand be evaluated by visualizing the graph and considering the "human-comprehensibility". Since this requires a ground truth, is prone to the subjective understanding and is usually performed for a few random samples, it is important to apply stable evaluation metrics. One relevant accuracy metric for synthetic datasets with ground truths is the Area Under the Receiver Operating Characteristic Curve (ROC-AUC) (see Richardson et al.[16]). The Receiver Operating Characteristic (ROC) curve plots the False Positive Rate (FPR) on the x-axis against the True Positive Rate (TPR), across different classification thresholds. The area under the curve (AUC) is calculated for said curve, resulting in the ROC-AUC. It is important to note, that a value of 0.5 equals random guessing, while a score of 1.0 indicates perfect classification.

TODO:
Variation in perturbation approaches lie in: mask gen. alg., type of mask, objective function.
Differentiate between "interpretable" and "explainable"? Model itself provides human-understandable interpretations vs model still black box with explanations by post-hoc model.
- Other metric includes fidelity, results of taxonomy propose only using PGExplainer for Node Classification as it achieves low fidelity on Graph tasks
TODO: figure of perturbation pipeline?

2.6 Boolean Satisfiability Problem

We define the Boolean Satisfiability Problem (SAT) according to Guo et al.[11][p.641]:
A Boolean formula is constructed from Boolean variables, that only evaluate to True (1) or False (0), and the three logic operators conjunction ($\wedge$), disjunction ($\vee$) and negation ($\neg$). SAT aims to evaluate whether there exists a variable assignment for a formula constructed of said parts so that it evaluates to True. If so, the formula is said to be satisfiable or unsatisfiable otherwise. Every propositional formula can be converted into an equivalent formula in conjunctive normal form (CNF), which consists of a conjunction of one or more clauses. These clauses must contain only disjunctions of at least one literal (a variable or its negation). In this work we consider only formulas in CNF, as NeuroSAT[19] assumes SAT problems to be in CNF. An example of a satisfiable formula in CNF over the set of variables $V = \{x_1, x_2\}$ is

$$\psi(V) = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_2)$$

with satisfying assignment $A : \{x_1 \mapsto 1, x_2 \mapsto 1\}$. Furthermore, SAT is NP-complete, meaning that if there exists a deterministic algorithm able to solve SAT in polynomial time, then such an algorithm exists for every NP problem (see cook[4]). Current state-of-the-art SAT solvers apply searching based methods such as Conflict Driven Clause Learning[15] or Stochastic Local Seach[18] with exponential worst-case complexity.

2.6.1 Representation as Bipartite Graph

SAT has extensively been studied in the form of graphs. Guo et al.[11] describe four different types of graph representations for CNF formulae with varying complexity and information compression. Since we want to minimize the loss of information for SAT we adapt the information-richest form of a literal-clause graph (LCG). A LCG is a bipartite graph that separates literals and clauses, with edges connecting literals to the clauses they appear in. The resulting graph can formally be described by a biadjacency matrix B of shape $l \times c$.

Let $A \in \mathbb{R}^{l+c \times l+c}$ be the adjacency matrix of our bipartite graph. Since for the bipartite case edges exist only between the two color classes l and c, the adjacency matrix can be represented as

$$A(i,j) = \begin{bmatrix} 0_{l\times l} & B \\ B^T & 0_{c\times c} \end{bmatrix}, \tag{2.24}$$

where 0 denotes a zero matrix in the shape of their subscript (see Sun et al.[20]).

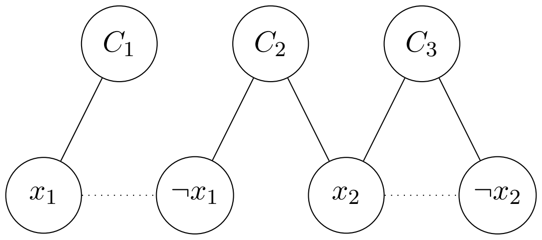

Figure 2.2: LCG representation of $\psi(V)$ with dashed lines representing the connection between complementary literals relevant for the message passing in GNNs.

2.6.2 Unsatisfiable Cores

The core of an unsatisfiable formula in CNF is a subset of the formula that is also unsatisfiable. Every unsatisfiable formula therefore is a core on its own, but can be broken down into smaller cores. The smaller a core the more significance it holds. A minimal unsatisfiable core is also referred to as a minimal unsatisfiable subset (MUS). SAT solvers like minisat[8] are able to compute unsatisfiable cores but do not generally provide a MUS due to high computational cost. However, several deletion-based algorithms exist for computing MUSs (see Torlak et al.[21]).

2.6.3 Backbones

Leave out for now.

2.7 TODO NeuroSAT

ML approach for SAT solving using message passing neural network. NeuroSAT[19]: Messages are passed between clauses and literals, as well as literals and their complement. 1. Clause receives from neighboring literals 2. Literals receive from clauses and complement. (Define flip function that swaps literal row with row of its negation; relevant for NeuroSAT)

Chapter 3

PGExplainer - Main part

V1: In the following chapter, we introduce the PGExplainer[**<empty citation>**] and its concepts. The idea is to generate explanations in the form of probabilistic graph generative models for any learned GNN model, henceforth referred to as the downstream task (DT), by utilizing a deep neural network to parameterize the generation process. This approach seeks to explain multiple instances collectively, as they share the same neural network parameters, and therefore improve the generalizability to previous works, particularly the GNNExplainer[**<empty citation>**].

V2: In the following chapter, we introduce the PGExplainer[**<empty citation>**] and its concepts. The idea is to generate explanations in the form of probabilistic graph generative models that have proven to learn the concise underlying structures of GNNs most relevant to their predictions. This approach may be applied to any learned GNN model, henceforth referred to as the downstream task (DT), by utilizing a deep neural network to parameterize the generation process. PGExplainer seeks to explain multiple instances collectively, as they share the same neural network parameters, and therefore improve the generalizability to previous works, particularly the GNNExplainer[**<empty citation>**]. This means that all edges in the dataset are predicted by the same model, which leads to a global understanding of the DT.

Transductive/Inductive relevant?

TODO: Maybe separate the theory of PGE from our own work more strictly? E.g. 3. PGE theory, 4. PGE reimplementation and NeuroSAT application?

We then describe our reimplementation in detail (Section 3.2), including the changes made and difficulties during the process.

In Section 3.3 we present the idea of applying PGExplainer on the NeuroSAT framework to generate explanations for the machine learning SAT-solving approach and comparing these to "human-understandable" concepts like UNSAT cores and backbone variables.

3.1 Theory

We follow the structure of the original Paper[**<empty citation>**] and start by describing the learning objective (Section 3.1.1), the reparameterization trick (Section 3.1.2), the global explanations (Section 3.1.3) and regularization terms (Section 3.1.4).

3.1.1 Learning Objective

3.1.2 Reparameterization Trick

The approach in PGExplainer is a common approach in ML for simplifying objectives? FIND LITERATURE THAT EXPLAINS APPROXIMATION OF COND. ENTROPY WITH CROSS ENTROPY. Explanation as simple as one formula for one graph variable example, cross entropy applied to whole distribution?

"We can modify the conditional entropy objective in Equation 4 with a cross entropy objective between the label class and the model prediction" (GNNExplainer)

3.1.3 Global Explanations

3.1.4 Regularization Terms

3.2 Reimplementation

Implementation details:

- Started by reimplementing the downstream tasks used in og paper for node and graph class.
- Node datasets taken from original and transformed to a "pyg format", to keep original structure and ground truths
- Graph: BA-2Motif from pyg library with self generated gt, MUTAG dataset taken from pyg and added gt-labels that were added in original dataset
- Created pytorch/pytorch geometric implementation of 3-layer graph conv networks
- architecture as described in paper: ReLu activation, Pooling for graph net
- Xavier initialization for all layers
- Same hyperparameters?(Adam, $1 * 10^-3$ lr, 1000 epochs)/experimental setup?

- ADDED Dropout(0.1) to improve performance/overfitting on node tasks
- Fully connected layer: torch geometric GraphConv to allow passing of edge weights(+ bipartite)!?
- Original references sageConv in paper, pyG impl. does not allow edge weights, verify!
- GATConv for Graph attention(referenced by original)
- Alternatives: GCNConv(edge weights, no bipartite graphs)
- 80/10/10 split
- =¿ Similar accuracies achieved

Explainer: This is irrelevant for paper, description of why reparametrization trick necessary - First approach tried passing a masked edge index to downstream task with edge weights ¿ 0.5 =¿ Unable to learn with hard cut-off (bad for gradients). (Same with TopK probably?!) - Pass calculated edge weights to downstream task for learning. If no edge weights are passed(downstream task), all edge weights are initialized with one to represent all edges beeing relevant

3.3 Application on NeuroSAT

Reimplementation of NeuroSAT provided by Rodhi. As the code used for NeuroSAT can also be found in our Repository, we stress that only the changes described in the following chapter are part of our work.

What did we do? What did we change for NeuroSAT? What data was used? How did we adapt PGExplainer?

Only change in NeuroSAT: pass edge weights into adjacency matrix. Calculates ....
Generated batches of unsat problems that "turned" unsat because of last added clause. 10 literals per problem. Only unsat to test for unsat cores, that only apply for unsat problems. Calculated unsat cores with solver xy by adding negative assumption literals per clause and passing these as assumption for calulation. The edges of the clauses present in the unsat core were treated as ground truth.

Changes for explainer: Edge embeddings calced by DS and passed to explainer. Calculated edge embeddings by concatenating node embeddings for connected nodes, similar to original. Embeddings fed into MLP, weights sampled with reparam. trick to get edge "probabilities", passed as "unbatched" sampled graph into NeuroSAT predictor. Visualization of SAT problems with edge weights, ands gts.

For quant. eval. adapted roc auc as metric as done in PGExplainer. Results seem ”good” but qual. eval. shows different result. roc auc bad metric?
For qual. eval. topk(=number of edges in gt) edges of predictions were highlighted to be compared to gt edges. For quant. eval. the edge probabilites were compared to gt with 1s for edges in gt and 0s for rest.

Chapter 4

Results

Experimental Setup: We follow the experimental setup from the PGExplainer as closely as possible. Since the textual description refers to the setup from GNNExplainer and is lacking in some aspects, we extract the missing information from the codebase. As the hyperparameters are unclear or not comprehensible for some tasks we also draw information from the configs of PYTORCH REIMPL.

We use normalization in our downstream models, though it is not described in the paper, since it is used in the code. We also experimented with the effects of the use of normalization since it seems to be relevant to the performance of the explainer.
The explainer is trained and evaluated on the same data. We also run experiments with added train/test splits TODO!
Not all data is fed into the explainer: BA-Shapes: BA-Community: Tree-Cycles: "First"/to base graph attached node of each motif in graph is used. Tree-Grid: All Motif nodes are used BA-2Motif: All graphs are used MUTAG: Only the graphs with an available ground truth are used. GT exist for mutagenic graphs that have either chemical groups NH2 or NO2. We later discuss if these selections make sense and run experiments with different data selections.
Quantitative: 10 explainer runs on one downstream model; Calculate ROC-AUC over ALL graphs/nodes in each run; Qualitative: Original uses a threshold; we instead take the topK nodes according to the dataset/motif as an explanation
We also discuss if treating the number of k edges as a hyperparameter dependant on the downstream task makes sense and propose having the network learn it, to improve generalizability and allow the explainer to work on data with varying size.

CPU vs GPU:
It is important to highlight that our code achieved better and way more stable results for BA-2Motif when trained on a gpu instead of cpu.
Ba-Shapes, Tree-Cycles and MUTAG results achieved were identical.

Ba-Community and Tree-Grid achieved very slightly better results on CPU.

Sweeps: (Params ordered by importance)
BA-Shapes: higher size reg -¿ 0.1; lower entropy reg -¿ 0.01; lr and tT very low impact but slightly higher -¿ 0.01 and 5. Note that Loss curve jumps on most runs! (logical-sweep-94 and restful-sweep-92 have clean loss) TRY HIGHER SIZE REG AND LOWER ENTROPY REG

BA-Community: lr 0.0001 too low, not working -¿ 0.003; lower entropy -¿ 0.1; higher size -¿ 0.1; TRY MORE SEEDS, LR, EPOCHS?

Tree-Cycles: high lr -¿ 0.01 ; lower entropy reg -¿ 0.1/0.01; higher size reg -¿ 0.1/0.01 ; lower tT -¿ 1. TRY WITH MORE SEEDS FOR ENT, SIZE, TEMP? Confirmed higher size reg -¿ 0.1; lower entropy reg -¿ 0.01; temp really low impact, tendency higher. TRY 30 EPOCHS???

Tree-Grid: high lr -¿ 0.01; high size reg -¿ 1; higher entropy reg? -¿ 10/1, high tT -¿ 5. TRY MORE SEEDS FOR ENTROPY REG -¿ not quite clear, tendency lower; MAYBE EVEN HIGHER LR -¿ No

BA-2Motif: RUN ON GPU - Not the cause. Cause for better results were features of 1 instead of 0.1! However, good results achieved on BA2-Motif dataset from pyg, not original one.
Comparison to orginal one: Original dataset transformed to pytorch performs way worse, for features of 0.1! Mean AUC of about 0.4!
Original dataset with features changed to ones instead of 0.1: Works good as well.
MUTAG: Low lr -¿ 0.0003; low entropy reg(high impact, but highest AUC runs vary) -¿ 0.1; low tT -¿ 1; less epochs -¿ 20; low size reg -¿ 0.005(/0.01); Loss is messy and AUC seems to decrease over time! lr 0.0001 worse, entropy reg 0.1/0.01 has zero effect -¿ 0.1

Effects of selected motif nodes for Node task: Compare Tree-Grid/Tree-Cycles performance when using all/one node per motif...

Chapter 5

Discussion

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Chapter 6

Conclusion

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

Bibliography

[1] A.S. Asratian, T.M.J. Denley, and R. Häggkvist. *Bipartite Graphs and their Applications*. Cambridge Tracts in Mathematics. Cambridge University Press, 1998. ISBN: 9781316582688. URL: `https://books.google.de/books?id=l8fLCgAAQBAJ`.

[2] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. "A comprehensive survey of graph embedding: Problems, techniques, and applications". In: *IEEE transactions on knowledge and data engineering* 30.9 (2018), pp. 1616–1637.

[3] Augustin Cauchy et al. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538.

[4] Stephen A Cook. "The complexity of theorem-proving procedures". In: *Logic, automata, and computational complexity: The works of Stephen A. Cook*. 2023, pp. 143–152.

[5] T.M. Cover and J.A. Thomas. "Entropy, Relative Entropy, and Mutual Information". In: *Elements of Information Theory*. John Wiley & Sons, Ltd, 2005, pp. 13–55. ISBN: 9780471748823. DOI: `https://doi.org/10.1002/047174882X.ch2`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/047174882X.ch2`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/047174882X.ch2`.

[6] Piotr Dabkowski and Yarin Gal. "Real time image saliency for black box classifiers". In: *Advances in neural information processing systems* 30 (2017).

[7] Reinhard Diestel. "Random Graphs". In: *Graph Theory*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 323–345. ISBN: 978-3-662-53622-3. DOI: `10.1007/978-3-662-53622-3_11`. URL: `https://doi.org/10.1007/978-3-662-53622-3_11`.

[8] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[9] Edgar N Gilbert. "Random graphs". In: *The Annals of Mathematical Statistics* 30.4 (1959), pp. 1141–1144.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[11] Wenxuan Guo et al. “Machine learning methods in solving the boolean satisfiability problem”. In: *Machine Intelligence Research* 20.5 (2023), pp. 640–655.

[12] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: `10.1109/5.726791`.

[13] Zhiyuan Liu and Jie Zhou. “Introduction”. In: *Introduction to Graph Neural Networks.* Cham: Springer International Publishing, 2020, pp. 1–3. ISBN: 978-3-031-01587-8. DOI: `10.1007/978-3-031-01587-8_1`. URL: `https://doi.org/10.1007/978-3-031-01587-8_1`.

[14] Dongsheng Luo et al. “Parameterized explainer for graph neural network”. In: *Advances in neural information processing systems* 33 (2020), pp. 19620–19631.

[15] Joao P Marques-Silva and Karem A Sakallah. “GRASP: A search algorithm for propositional satisfiability”. In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521.

[16] Eve Richardson et al. “The receiver operating characteristic curve accurately assesses imbalanced datasets”. In: *Patterns* 5.6 (2024), p. 100994. ISSN: 2666-3899. DOI: `https://doi.org/10.1016/j.patter.2024.100994`. URL: `https://www.sciencedirect.com/science/article/pii/S2666389924001090`.

[17] Franco Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: `10.1109/TNN.2008.2005605`.

[18] Bart Selman, Henry A Kautz, Bram Cohen, et al. “Local search strategies for satisfiability testing.” In: *Cliques, coloring, and satisfiability* 26 (1993), pp. 521–532.

[19] Daniel Selsam et al. “Learning a SAT solver from single-bit supervision”. In: *arXiv preprint arXiv:1802.03685* (2018).

[20] Hanqing Sun et al. “Supervised biadjacency networks for stereo matching”. In: *Multimedia Tools and Applications* 83 (June 2023), pp. 1–26. DOI: `10.1007/s11042-023-15362-5`.

[21] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. “Finding Minimal Unsatisfiable Cores of Declarative Specifications”. In: *FM 2008: Formal Methods.* Ed. by Jorge Cuellar, Tom Maibaum, and Kaisa Sere. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 326–341. ISBN: 978-3-540-68237-0.

[22] Hao Yuan et al. “Explainability in graph neural networks: A taxonomic survey”. In: *IEEE transactions on pattern analysis and machine intelligence* 45.5 (2022), pp. 5782–5799.

Eidesstattliche Versicherung

(Affidavit)

Name, Vorname (surname, first name)	Matrikelnummer (student ID number)

☐ Bachelorarbeit (Bachelor's thesis)   ☐ Masterarbeit (Master's thesis)

Titel
(Title)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.	I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Ort, Datum (place, date)	Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Ort, Datum (place, date)	Unterschrift (signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**