POLITECNICO DI TORINO

Master's Degree in
Communications and Computer Networks Engineering

Master's Degree Thesis

Applications of Machine Learning Techniques to Graph Neural Networks: Design of an Algorithm for the Maximum Independent Set and Minimum Vertex Cover Problems

Supervisors

Prof. Giulia FRACASTORO

Prof. Sophie FOSSON

Dr. Grigoris CHRYSOS

Prof. Volkan CEVHER

Candidate

Lorenzo BRUSCA

October 2023

Abstract

This study introduces an innovative framework for addressing the maximum independent set (MIS) and minimum vertex cover (MVC) problems using a graph neural network (GNN) approach inspired by dynamic programming (DP). The MIS problem finds important applications in Communication Networks, as it can help in optimizing the network's performance by ensuring non-interfering or non-conflicting communication between nodes. Instead, identifying an MVC in a network helps to minimize the number of resources needed for infrastructure deployment, which is crucial in applications like transportation or utility networks. Unfortunately, the MIS and MVC are NP-hard problems, and classic heuristics, like the simplex or the ellipsoid methods, may take too much time to find a solution. For this reason, in the last few years, new methods involving Machine Learning (ML) models have been deployed. Since the MIS and MVC problems are NP-hard, producing labels to train the ML-like approaches would be too difficult, thus unsupervised training is employed. If the ML models are properly trained, these approaches have the advantage of being faster than standard heuristics, as the complexity of the method is linked to the complexity of the ML model architecture. Unfortunately, ML-like approaches may lead to solutions of the MIS and MVC problems that are too far from the optimal value. The goal of this work is to produce a new algorithm that, by making use of an ML model, shares the low complexity of standard ML-like methods but outperforms them by making use of an ML model that is trained self-supervised. Moreover, the architecture of the proposed ML model relies on Graph Neural Networks (GNNs), that leverage the concept of embeddings in order to capture diverse semantic meanings of graphs.

The proposed algorithm employs a DP-like recursive structure. Each iteration of the algorithm considers a graph as input and outputs a graph of lower complexity (thus with fewer nodes or fewer edges) that is employed in the next iteration. The first step of each iteration consists of creating two smaller sub-graphs. Both sub-graphs have a lower complexity than the graph from which they were generated. Then, the second step is executed by an ML model that acts as a binary classifier, indicating which of the two graphs is chosen. Finally, in the third step, a check over the chosen graph is made. If the chosen graph has a specific structure, the algorithm stops, otherwise the chosen graph is used in the next iteration. To train the ML model effectively, annotated comparisons between different graphs in terms of their MIS and MVC sizes are leveraged. By annotating these comparisons with the outcomes generated by the algorithm, a self-training process is established that results in improved self-annotation of the comparisons and vice versa. Theoretical results

prove that if the ML model is effectively trained, then the proposed algorithm leads to solutions that are near-optimal. Finally, numerical results show that the proposed method outperforms the state-of-the-art approaches across various synthetic and real-world datasets.

Acknowledgements

Un primo ringraziamento va alle Professoresse Giulia Fracastoro e Sophie Fosson, per il supporto fornito durante la stesura della tesi. I miei più sentiti ringraziamenti anche a Grigoris e Stratis, che mi hanno enormemente aiutato a sviluppare questo lavoro e a scrivere l'articolo che deriva da questa tesi.

Mi sono sempre chiesto come si potessero scrivere dei ringraziamenti che andassero a toccare nel profondo tutte le persone più importanti della mia vita. E ho pensato che la cosa migliore è ringraziare tutti per la cosa che più mi lega a loro, l'amore. L'amore che è di vari tipi, come l'amore che si prova per un amico e che è quello che provo per i miei amici di Roma, per i miei amici di Torino e per quelli di Losanna. Oppure l'amore che mi lega ai miei familiari, ad esempio quello per i miei zii, per nonna e per mio fratello Alessandro. Molto diverso ma molto forte è l'amore che si prova per la propria ragazza, e per questo ringrazio Ila, la persona che da 5 anni mi sta accanto e mi guida e aiuta in ogni scelta che prendo. Infine, vorrei chiudere con l'amore che si prova per i propri genitori, e che mi lega da sempre alla mia mamma e al mio papà. Due incredibili persone che oltre ad avermi cresciuto e appoggiato sempre nella mia vita, mi hanno insegnato che la cosa più importante e più bella che ci sia è proprio quella per cui sto ringraziando tutti quanti, l'amore.

Table of Contents

List of Tables

List of Figures

Acronyms

MIS

Maximum Independent Set

MVC

Minimum Vertex Cover

ML

Machine Learning

GNN

Graph Neural Network

DP

Dynamic Programming

MCTS

Markov Chain Tree Search

CO

Combinatorial Optimization

GEM

Graph Embedding Module

FCNN

Fully Connected Neural Network

Chapter 1

Introduction

An unprecedented success has been achieved by deep neural networks (DNNs) in extracting intricate patterns directly from data without the need for handcrafted rules, while still generalizing well to new and previously unseen instances [1, 2]. Among the several fields of application, in the last few years, Combinatorial Optimization (CO) problems have been successfully tackled by DNNs, such as the Traveling Salesman Problem [3, 4, 5], the Job-Shop Scheduling Problem [6, 7], and the Quadratic Assignment Problem [8].

A fundamental obstacle faced by DNNs techniques when applied to CO problems is the scarcity of training data. Generating annotations for this data entails solving a vast number of instances of CO, which makes supervised learning approaches computationally impractical for NP-hard problems, as highlighted in [9]. Overcoming this challenge is crucial to realizing the complete potential of deep neural networks (DNNs), which otherwise have wide-ranging utility in the domain of CO.

DNNs are not the only tool for solving CO tasks. Several works have used dynamic programming (DP) approaches in order to break the initial complexity of a CO problem into several less complex instances. The Vehicle Routing Problem [10], the Knapsack Problem [11], and the Graph Coloring Problem [12] are only a few examples of CO problems tackled with DP-like frameworks. The main limitations of DP-like algorithms are in terms of time and space. As DP focuses on breaking down a problem into several sub-problems, this operation can be time-limiting, as every sub-problem can potentially produce several different sub-sub-problems and so on. Moreover, if the number of sub-tasks is too high, the memory required to store all of them may be too big. Circumventing these problems, by properly changing the

inherent structure of a classical DP algorithm, can lead to an algorithm that is fast and efficient.

This work, resulted in a paper accepted at the NeurIPS conference[1], consists of solving two classical CO problems applied to graphs: the *Maximum Independent Set* (MIS) and the *Minimum Vertex Cover* (MVC) problems. The MIS consists of finding the *independent set* (IS) of the biggest size, where an IS is a set of nodes such that every couple of nodes in the IS is not connected with an edge. The MVC goal is, instead, to find the set of nodes (defined as *vertex cover* (VC)) of minimum cardinality such that every edge of the graph has at least one of the two nodes defining it in the set.

This work proposes two novel algorithms, sharing the same structure, for solving the MIS and MVC problems. The algorithms leverage the capabilities of a *Graph Neural Network* (GNN) model [13]. GNNs have been used in the last few years for their ability to learn representations of graph-structure data independently on the size of the graph [14]. They can effectively capture into vector representations, known as *embeddings*, the patterns and semantics of a graph.

The proposed algorithms share a DP-like structure. Given a graph as input, the algorithms iteratively break a graph into two sub-graphs of lower complexity. This procedure is followed until one of the sub-graphs assumes a specific structure. The novelty brought by the proposed methods lies in the division, or branching, process. A model composed of GNNs decides which of the two sub-graphs will be used in the next iteration, and the un-chosen sub-graph is no longer taken into consideration. By following this approach, the time and space limitations that are inherent to classical DP-like methods are drastically reduced, as the number of sub-tasks is limited by the GNN-like model.

If on one side the GNN-like model lightens the complexity of the DP-like proposed algorithms, on the other side the DP-like structure helps the model by producing training data, even if the MIS and MVC problems are known to be NP-hard. Indeed, this work proposes a self-supervised learning approach where the DP-like algorithms produce the ensemble of training data and labels that are necessary for efficiently training the GNN-like model.

[1] *Maximum independent set: Self-training through dynamic programming.* Lorenzo Brusca, Lars C.P.M. Quaedvlieg, Stratis Skoulakis, Grigorios Chrysos and Volkan Cevher. 37th Conference on Neural Information Processing Systems (NeurIPS 2023)

A precise validation of the proposed methods is conducted in diverse graph distribution datasets. The proposed DP-like algorithms show better performances than previous DNN methods, like [15, 16, 17, 18], in almost all datasets.

1.1 Thesis Outline

The structure of this work is organized in 8 chapters, organized as follows:

- In Chapter 1 the introduction of this work is provided.
- Chapter 2 presents a description of all topics covered by this work.
- Chapter 3 delves into the analysis of the proposed algorithms and introduces the important *Comparator Function*.
- In Chapter 4 the architecture of the GNN-like model is analyzed.
- Chapter 5 details the self-training process, describing both the crucial *Consistency Property* and the data generation process.
- Chapter 6 focuses on the ensemble of modifications to the standard methods that are carried out to improve the final results.
- In Chapter 7 the performances of both the standard proposed approaches and the modified approaches are compared against diverse baseline methods.
- Finally, Chapter 8 details the conclusions and future changes to improve the proposed methods.

Chapter 2

Background

This chapter provides an overview of the ensemble of topics that are treated in this work. In particular, these are the topics covered by the sections of this chapter:

- Section 2.1 defines a standard CO problem and the two problems analyzed in this work.
- Section 2.2 presents a general explanation of classic DP and MCTS methods.
- Section 2.3 provides an overview of general ML learning approaches for training an ML model.
- Section 2.4 describes the structure of Recurrent GNNs.
- Section 2.5 shows several Convolutional GNNs methods.

2.1 Combinatorial optimization problems over graphs

In this section, an analysis of CO problems over graphs is conducted. In particular, a general overview of graphs and CO problems over graphs is given, followed by a description and formulation of typical CO graph problems.

2.1.1 Graph definition and notation

The following paragraphs show the graph notations that are employed in all the subsequent chapters of this work.

Graph definition. A graph $G(V, E)$ is an ordered couple that describes relations between different entities. Such entities are called nodes or vertices, while the relations between nodes are called edges or arcs. The set of all nodes is denoted as V, while the set of all edges is denoted as E. Every edge is defined as a couple of nodes, thus E is a subset of $V \times V$ (where $\times$ denotes the cartesian product), namely $E \subseteq \{(u, v)|u, v \in V\}$. Every edge (u, v) can be assigned a weight $w_{(u,v)}$, which is a number that can model the strength of the relation among the two nodes u, v defining the edge.

Graph types. If the couples of nodes defining the edges are ordered couples, the graph is referred to as *directed*, otherwise, the graph is *undirected*. Moreover, a graph is *permitting-loops* if admits edges (u, u), connecting a node to itself. Vice versa, the graph is *permitting-no-loops*. Lastly, *unweighted* graphs admit only one possible weight for every edge, contrary to the *weighted* graphs.

Graph features. The set of weights of a graph G are generally gathered into a square matrix $\mathbf{A}_G$ denoted as the *Adjacency matrix*. Every element $\mathbf{A}_G(i, j)$ is equal to $w_{i,j}$ if the edge (i, j) exists, otherwise it is equal to 0. In the case of an undirected graph, the matrix is symmetric, since $w_{i,j} = w_{j,i}$. The *neighbors* of a node v is the set of nodes that are directly connected through an edge with v, namely $ne[v] = \mathcal{N}(v) = \{u \in V \mid (u, v) \in E\}$, while $co[v]$ denotes the set of arcs having v as a node. The *degree* of a node v is the total number of nodes connected to v, namely $d(v) = |ne[v]| = |\mathcal{N}(v)|$. Moreover, the degrees of a graph are generally gathered into a diagonal matrix $\mathbf{D}$. For every node couple (i, j), the element $\mathbf{D}(i, j)$ is equal to 0 if $i \neq j$ and it is equal to $d(i)$ if $i = j$.

2.1.2 Combinatorial optimization: a brief summary

Combinatorial optimization is a branch of optimization that focuses on finding the best possible solution among a finite set of discrete options or combinations. It involves selecting an optimal arrangement or combination of elements from a given set to optimize a defined objective function while satisfying specific constraints.

A CO problem is defined by a set of discrete *decision variables* $\mathbf{x} = (x_i)_{1 \leq i \leq N} \in S$ and a *objective function*, sometimes denoted as *cost function* or *energy function*, $f(\mathbf{x}) : S \to \mathbb{R}$, that maps the decision variables into a real number. The set $S \in \mathbb{R}^N$ is named as the *solution space* and contains the possible values that the decision

variables can assume. The goal of the problem is to find a value $\overline{\mathbf{x}} \in S$ such that f is either maximized or minimized under a set of *constraints*. Maximization and minimization are equivalent, since minimizing $f(\mathbf{x})$ is totally equivalent to maximizing $-f(\mathbf{x})$.

Some CO problems admit a *Linear programming* (LP) formulation, meaning that the objective function and the constraints can be expressed as a linear combination of the decision variables. Moreover, if the decision variables assume only integer values, then the formulation is said to be *Integer*-linear programming (ILP). In the latter case, the ILP problem is typically formulated with the following canonical form:

$$\begin{aligned} &\text{maximize} && \mathbf{c}^T\mathbf{x}, \\ &\text{subject to} && \mathbf{A}\mathbf{x} \leq \mathbf{b}, \\ &\text{and} && \mathbf{x} \geq 0. \end{aligned} \tag{2.1}$$

In the above formulation, $\mathbf{x}$ is the vector of decision variables, $\mathbf{c}$ is the vector of the cost function coefficients, $\mathbf{b}$ is the constraint vector and $\mathbf{A}$ is the constraint matrix. The above formulation is used later to define the problems analyzed in this work.

2.1.3 Case study

Within the class of CO problems, a specific category is dedicated to addressing problems formulated on graph structures. In this work, two CO problems are examined: the MIS and the MVC. The subsequent sections provide succinct explanations and descriptions of these problems.

Maximum independent set problem

In the context of a graph $G(V, E)$, an independent set is defined as a set of nodes $S \subseteq V$ where, for any two vertices in S, there is no edge connecting them. In other words, for any nodes u and v belonging to S, the edge (u, v) does not exist in E. The objective of this problem is to identify the independent set with the maximum number of nodes, denoted as MIS(G) in the current study.

As well as other notorious graph theory problems, the MIS admits an ILP formulation. Particularly, given a graph $G(V, E)$ with $n = |V|$, the binary decision variables are $\mathbf{x} = (x_i)_{1 \leq i \leq n} \in \{0,1\}^n$ for each vertex $i \in V$ such that $x_i = 1$ if vertex i is considered as part of the solution and $x_i = 0$ otherwise.

In MIS, the energy function is defined as the sum of every binary decision variable: $F(G, \mathbf{x}) = \sum_{i \in V} x_i$. Since every edge (i, j) can have at most one of its nodes in the independent set (otherwise the independent condition is violated), the sum of

decision variables of i and j is at most one:

$$\begin{aligned} &\text{maximize} && \sum_{i \in V} x_i, \\ &\text{subject to} && x_i + x_j \leq 1 \quad \forall\, (i,j) \in E, \\ &\text{and} && x_i \in \{0,1\} \quad \forall\, i \in V\ . \end{aligned} \tag{2.2}$$

Minimum vertex cover problem

A vertex cover is a set of vertices $S \subseteq V$ such that, for every edge $(u, v) \in E$, at least one of the vertices u or v is in S. The MVC problem focuses on finding the vertex cover with the smallest cardinality, denoted as MVC(G).

Adopting the notation used for the MIS problem, the ILP formulation is employed for the MVC problem. In contrast to the MIS case, the objective function $F(G, \mathbf{x}) = \sum_{i \in V} x_i$, is minimized. Moreover, due to the requirement that each edge of the graph must be covered, it follows that the sum of the decision variables corresponding to the two nodes defining an edge must be greater than or equal to one:

$$\begin{aligned} &\text{minimize} && \sum_{i \in V} x_i, \\ &\text{subject to} && x_i + x_j \geq 1 \quad \forall\, (i,j) \in E, \\ &\text{and} && x_i \in \{0,1\} \quad \forall\, i \in V\ . \end{aligned} \tag{2.3}$$

2.1.4 NP-hardness of MIS and MVC: so what?

As said before, the MIS and MVC problems can be solved by using ILP. Several algorithms have been proposed for solving linear programs, like the *interior point method* [19], *branch and bound* [20], or the *cutting planes method* [21]. In this section, two famous methods are briefly described, along with an analysis of their complexity.

Simplex method

The simplex method [22] operates based on the fundamental concept that the feasible solutions of a linear program can be represented by a polytope, a convex body in n-dimensional space with flat faces. These faces are determined by the set of constraint equations $\mathbf{a}^T\mathbf{x} \leq \mathbf{b}$, where each equation divides the space in half along the hyperplane $\mathbf{x}|\mathbf{a}^T\mathbf{x} = \mathbf{b}$. The objective of the linear program is to find the corner of the polytope that maximizes the objective function $\mathbf{c}^T\mathbf{x}$. To achieve this, the simplex method begins at any corner and systematically traverses

along the edges, continuously moving towards a higher-valued objective function until reaching an optimal corner. At each step, the current location is represented by a set of n constraints (called the basis) in the form $\mathbf{a}^T\mathbf{x} = \mathbf{b}$, which uniquely identifies a corner as it can be solved by solving a system of n equations with n unknowns. Then, one of the constraints is swapped for one of the constraints in the basis (this is called *pivoting* process) and a new corner is reached. In the worst case, every corner is visited and the total number of steps taken by the algorithm is 2^N. However, if the pivoting is performed smartly enough [23], then the number of steps taken by the algorithm is $\mathcal{O}(N)$, each costing $\mathcal{O}(N^2)$. So, in total, the complexity is $\mathcal{O}(N^3)$, as shown by Table 2.1.

Ellipsoid method

The ellipsoid method [24] aims to generate a sequence of ellipsoids $\mathcal{E}^0, \cdots, \mathcal{E}^i, \cdots$ with a dimensionality N (the same as the dimensionality of the solution space S mentioned in Section 2.1.2). The idea behind the method is to progressively reduce the volume of the ellipsoid until a possible solution is found. Given an ellipsoid $\mathcal{E}^i$, the sub-gradient ($\mathbf{g}^i$) of the cost function f is computed at the center $\mathbf{z}_i$ of the ellipsoid. Then, the half-plane $\mathbf{g}^{(i)T}(\mathbf{z} - \mathbf{z}^{(i)})$ splits the ellipsoid into two halves, and the subsequent ellipsoid in the sequence, $\mathcal{E}^{i+i}$, is the smallest ellipsoid that encloses the half-ellipsoid $\mathcal{E}^{(i)} \cap \mathbf{z}|\mathbf{g}^{(i)T}(\mathbf{z} - \mathbf{z}^{(i)}) \geq 0$. If $\mathcal{E}^{(i)}$ contains the solution of the problem, then also $\mathcal{E}^{(i)}$ is guaranteed to enclose the maximizer of f. This process continues until an ellipsoid with a sufficiently small volume is attained. Table 2.1 shows that the ellipsoid method exhibits polynomial worst-case complexity, unlike the simplex method which has an exponential worst-case complexity. However, the higher typical complexity of the ellipsoid method makes the simplex method to be the preferred choice in practice.

About the complexity of MIS and MVC

The MIS and MVC problems can be solved using either the simplex or the ellipsoid method since they admit an ILP formulation. Unfortunately, as highlighted by Table 2.1, the typical computational complexity for the ellipsoid method is polynomial of order 5, which can be a problem for graphs of big size, and it is even exponential in the worst case for the simplex method! For this reason, there is a need for new tools and algorithms to solve more efficiently such problems. Deep learning can help in this direction by drastically reducing the complexity needed for solving the aforementioned graph theory problems.

Table 2.1: Computational complexity of the simplex and ellipsoid method. The complexity is expressed as a function of the dimensionality of the decision variables. The following values are copied from the work of [23].

Method	Typical	Worst
Simplex	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2 2^N)$
Ellipsoid	$\mathcal{O}(N^5)$	$\mathcal{O}(N^8)$

2.2 Dynamic programming and MCTS

The algorithm that is designed in this work to solve the MIS and MVC problems leverages *Dynamic Programming* (DP) techniques. Moreover, the *tree search* structure that is explained in the following chapters strongly resembles the architecture of the *Markov Chain Tree Search* (MCTS). For these reasons, a short explanation of these two approaches is conducted.

2.2.1 Dynamic Programming

Invented in the 1950s by Richard Bellman, DP [25] is a technique that aims at finding a solution to a problem in polynomial time, and not exponential as for other methods (e.g. the worst-case complexity of the simplex method, Section 2.1.4). DP relies on the idea of dividing a complex problem into smaller sub-problems and then solving the latter ones. In order to do that, the algorithm is built on two principles: *Optimal substructure* and *Overlapping sub-problems*.

Optimal substructure
The optimal substructure property enables the reduction of problem complexity by decomposing it into smaller sub-problems. To compute the solution to the original problem, the solutions to the sub-problems are computed and then combined to determine the overall solution. This approach is advantageous because combining the sub-problem solutions is easier than directly calculating the solution to the original problem, as the sub-problems have lower complexity.

Overlapping sub-problems As a problem is broken down into sub-problems using the optimal substructure property, the same sub-problems are frequently encountered. This property, known as the overlapping sub-problems property, suggests that storing the solutions to these sub-problems in a table, often referred to as the *look-up table*, is a common and efficient approach. By consulting the look-up table, it is possible to quickly determine if a solution to a sub-problem has already been calculated when encountering it again.

2.2.2 Monte Carlo Tree Search

The Monte Carlo tree search method [26] is a search algorithm used in decision-making processes for complex problems, especially in areas such as game-playing, planning, and optimization. It uses a tree structure, known as the game tree or search tree, to represent the possible states and actions of the game being played. The basic idea behind MCTS is to simulate multiple random plays or "rollouts" in order to estimate the value of different actions or moves from a given state. Then, the value of the estimates is propagated to the nodes of the tree, in order to update the current node values. The following four steps, visually shown by Figure 2.1, summarize the MCTS algorithm:

- Selection: Starting from the root of the tree, the MCTS method traverses the tree by selecting child nodes based on a selection policy until it reaches a leaf node.

- Expansion: Once a leaf node is reached, the MCTS algorithm expands it by adding one or more child nodes representing possible actions or moves from that state. These new nodes are added to the search tree.

- Simulation (or Rollout): the MCTS method performs a Monte Carlo simulation or rollout from the newly added node(s). It plays out the game from the expanded state by making random moves until reaching a terminal state (e.g., end of the game). The result of this random play is used to estimate the value or quality of the newly added nodes.

- Backpropagation: After the simulation is complete, the algorithm backpropagates the results up the tree, updating the value estimates of all nodes along the path from the expanded node to the root. This information helps guide future selections and influence the overall search strategy.

2.3 Training approaches in Machine Learning

In this work, the model employed for solving the graph theory problems defined in Section 2.1.3 is trained with a self-supervised approach. Since it is not a standard approach, an analysis of this technique is conducted, highlighting the difference with respect to well-known training techniques, like the supervised approach.

Supervised learning
Supervised Learning (SL) is based on the fundamental concept of providing a system with input examples and their corresponding desired outputs, often referred

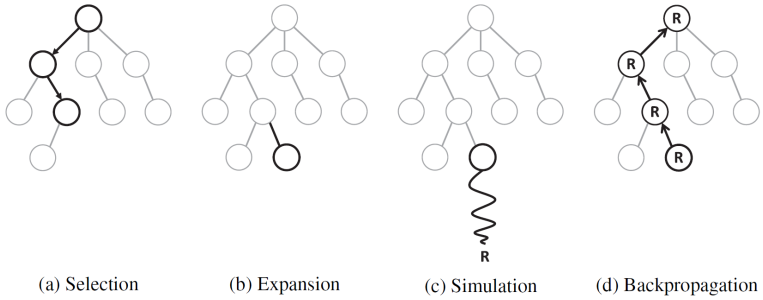

Figure 2.1: The image shows multiple steps of the Monte Carlo tree search algorithm. Each step generates a new state and a new estimate, that is backpropagated to update the state values.

to as labels. The primary objective of SL is to acquire a generalized rule or pattern that facilitates the mapping of new inputs to their respective outputs. In essence, SL represents a structured learning process that operates under the guidance of provided examples in a controlled manner.
Going into further detail, the SL learning method constructs a mathematical model by utilizing a dataset that comprises both input data and corresponding labels. This dataset, commonly referred to as training data, includes a collection of specific instances used for training purposes. Through an iterative optimization process, driven by a defined cost function, the SL method learns a function that facilitates the prediction of output values for new, unseen inputs. The ultimate aim is to establish an optimal function that accurately predicts outputs for inputs not encountered during training. This ability to perform accurately on previously unseen data, known as test data, is a key measure of success for any machine learning model. As an algorithm gradually improves the accuracy of its outputs over time, it is considered to have acquired the task and is expected to demonstrate high accuracy in handling unlabeled examples encountered in real-world scenarios.

Unsupervised Learning
Unsupervised learning (UL) is a machine learning approach where the learning algorithm is trained on raw, unlabeled data without any specific guidance or explicit feedback. Unlike supervised learning, unsupervised learning does not rely on pre-labeled examples to learn patterns or make predictions. Instead, it focuses on extracting meaningful information, identifying patterns, and finding underlying structures within the data.
The goal of unsupervised learning is to uncover hidden relationships, clusters, or structures in the data without prior knowledge or labeled information. It allows the method to autonomously explore and discover patterns that may not be immediately apparent. Unsupervised learning can be seen as a form of exploratory analysis, as

it enables researchers to gain insights and knowledge about the data, often leading to further investigations or more focused analysis.

Reinforcement Learning
Reinforcement learning (RL) involves an agent learning to interact with an environment to maximize a reward signal. Unlike supervised and unsupervised learning, reinforcement learning operates in a dynamic and sequential decision-making setting.
In reinforcement learning, an agent learns through a process of trial and error by taking actions in an environment and observing the resulting states and rewards. The agent aims to learn an optimal policy, which is a mapping from states to actions, that maximizes the cumulative reward over time. The agent explores the environment by taking actions and receives feedback in the form of rewards or penalties based on the desirability of its actions.

Semi-supervised Learning
Semi-supervised learning (Semi-SL) integrates concepts from SL and UL. It involves a dataset that includes both labeled and unlabeled data, and the model is trained on both types of data. Initially, the model is trained using the labeled data in a supervised manner. Subsequently, the model generates labels for the unlabeled data through its own predictions. If the model's label generation process demonstrates sufficient confidence, these generated labels are then utilized to further train the model.

Self-supervised Learning
In self-supervised learning (Self-SL), the model itself generates labels after each iteration. These labels, generated based on the current model parameters, are then utilized to train the model in the next iteration. This iterative process allows the model to improve its performance by continually updating the labels and adjusting its parameters accordingly. Furthermore, as the training progresses and the model is exposed to more data samples, the accuracy of label predictions improves. This improvement is due to the model's parameters being refined and adjusted based on the increasing amount of training data.
Self-supervised learning utilizes unlabeled data, which is advantageous as it enables learning from abundant unlabeled datasets without the need for external annotations. However, self-supervised learning generally requires longer training times compared to other learning methods.

2.4 Recurrent Graph Neural Networks

Graph Neural Networks (GNNs) have gained significant traction in recent years since their introduction by Scarselli [27] in 2009. They have found successful applications in various fields, like Communication Networks [28], Bioinformatics [29], and Natural Language Processing [30]. In this work as well, GNNs are employed, specifically for generating what is known as *embeddings*. The embeddings are low-dimensional vectors that capture crucial characteristics or features of the corresponding graph elements. In particular, node embeddings and edge embeddings encode information about individual nodes and edges within the graph, respectively. On the other hand, graph embeddings capture collective information and characteristics of the entire graph structure.

GNNs can be broadly categorized into two groups: *Recurrent GNN* (RecGNN) and *Convolutional GNN* (ConvGNN). In this section, the first group, RecGNN, is analyzed while the discussion on ConvGNN is reserved for Section 2.5. For the description of RecGNNs, the approach presented in the work by Scarselli [27] is followed, while the description of ConvGNNs is based on the work by Wu [13]. It is worth noting that these two works employ different notations, leading to the use of distinct notations for Section 2.4 and Section 2.5.

2.4.1 Labels and notations

In addition to the notation introduced in Section 2.1.1 for a graph $G(V, E)$, it is assumed that each node $v \in V$ and each edge $(v, u) \in E$ is associated with a label $\mathbf{l}_v \in \mathbb{R}^{l_V}$ and a label $\mathbf{l}_{(v,u)} \in \mathbb{R}^{l_E}$, respectively. The node labels represent features of the objects that the nodes represent, while the edge labels indicate the relationships between two objects. For instance, in the context of a social network graph where nodes represent individuals, the node labels may convey characteristics of the person represented by each node, and the edge labels may reflect the strength of friendships between different individuals.

In the following section is considered a supervised learning framework, aimed at training a model to find the embedding of every node of a graph. To this end, the following dataset is employed:

$$\begin{aligned}\mathcal{L} = \{(G_i, v_{i,j}, \mathbf{t}_{i,j})|, G_i = (V_i, E_i) \in \mathcal{G}, v_{i,j} \in V_i; \\ \mathbf{t}_{i,j} \in \mathbb{R}^m, 1 \leq i \leq p, 1 \leq j \leq N_i\}.\end{aligned} \tag{2.4}$$

Every datasample of the dataset is a triple, composed by a graph G_i, a node $v_{i,j}$ that is part of the nodes of G_i, and a target $\mathbf{t}_{i,j}$, where the target corresponds to

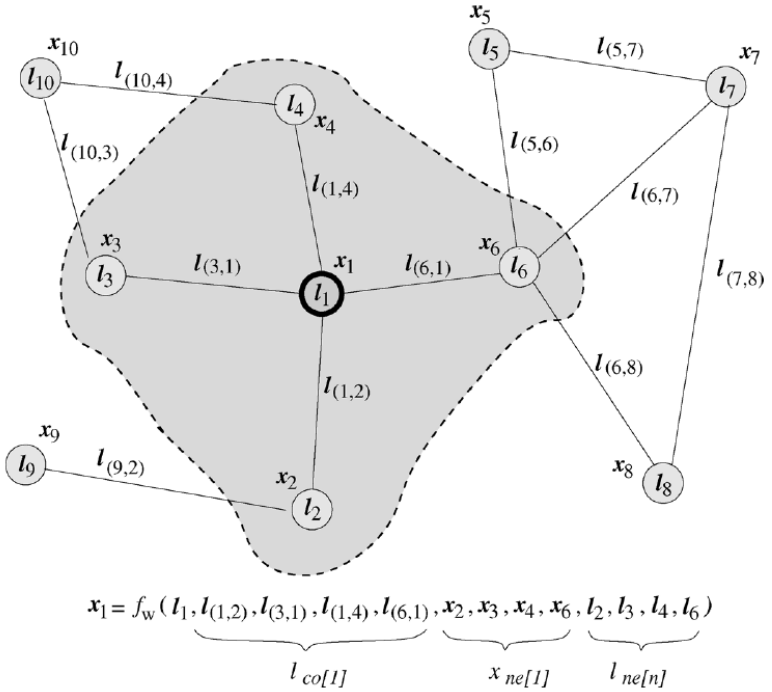

Figure 2.2: The picture shows a graph along with the node labels and edge labels. The circles indicate the presence of a node. The grey area indicates the set of labels and states that influence the computation of the state $\mathbf{x}_1$ of node $v = 1$.

the embedding of the node. Moreover, it is assumed that p graphs are used for training and that each graph G_i has N_i nodes.

2.4.2 Model: states and parametric functions

Each node v is assigned a state $\mathbf{x}_v$. The state $\mathbf{x}_v \in \mathbb{R}^S$ captures information not only about the node itself but also about the edges connecting the node and its neighboring nodes, as depicted in Figure 2.2. The relationship between $\mathbf{x}_v$ and its neighbors, edges, and itself is described by a parametric function $f_{\mathbf{w}}$, referred to as the *local transition function*. Furthermore, $\mathbf{x}_v$ is utilized to compute the output $\mathbf{o}_v$, which represents the decision pertaining to the node. The function that expresses the dependence of $\mathbf{o}_v$ on the current state and label of the node is known as the *local output function*. The following equations provide a definition for the local transition and local output functions:

$$\begin{aligned} \mathbf{x}_v &= f_{\mathbf{w}}(\mathbf{l}_v, \mathbf{l}_{co[v]}, \mathbf{x}_{ne[v]}, \mathbf{l}_{ne[v]}), \\ \mathbf{o}_v &= g_{\mathbf{w}}(\mathbf{x}_v, \mathbf{l}_v). \end{aligned} \tag{2.5}$$

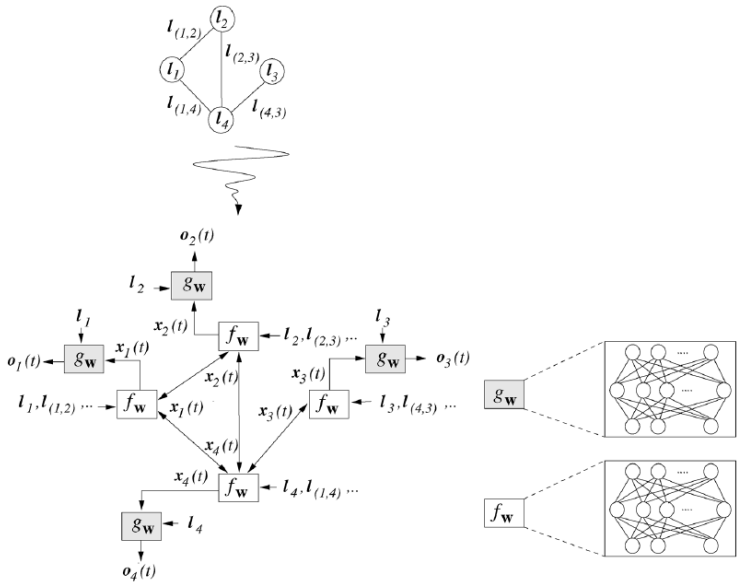

Figure 2.3: On the top, the graph is displayed. In the middle, every graph node is replaced by the two units $f_{\mathbf{w}}$ and $g_{\mathbf{w}}$. The two units can be implemented by feed-forward networks, as shown by the picture in the middle-right part of the figure.

In general, a condensed form of the previous equation is employed by stacking the states, outputs, labels, and node labels into vectors $\mathbf{x}$, $\mathbf{o}$, $\mathbf{l}$, and $\mathbf{l}_V$, respectively:

$$\begin{aligned} \mathbf{x} &= F_{\mathbf{w}}(\mathbf{x}, \mathbf{l}), \\ \mathbf{o} &= G_{\mathbf{w}}(\mathbf{x}, \mathbf{l}_V). \end{aligned} \tag{2.6}$$

In Equation 2.6, the functions $F_{\mathbf{w}}$ and $G_{\mathbf{w}}$ are referred to as the *global transition function* and the *global output function*, respectively.

The objective is to establish a mapping $\phi_{\mathbf{w}} : \mathcal{D} \rightarrow \mathbb{R}^m$ that assigns an output $\mathbf{o}_v$ to each node v in a graph, representing the node's embedding. Consequently, Equation 2.6 has to be solved, and in order to accomplish this, the implications of Banach's fixed-point (BFP) theorem [31] are leveraged. According to the theorem, if $F_{\mathbf{w}}$ from Equation 2.6 is a constructive mapping, the following conditions are satisfied:

1. There exists a solution to Equation 2.6, and such solution is unique.

2. The unique solution can be found by the Jacobi iterative method [32], namely:

$$\mathbf{x}(t+1) = F_{\mathbf{w}}(\mathbf{x}(t), \mathbf{l}). \tag{2.7}$$

3. The dynamical system in Equation 2.7 converges to the solution exponentially fast given any initial state value $\mathbf{x}(0)$.

Thanks to the BFP theorem, it is possible to iteratively solve Equation 2.5 for every node v by using the following set of formulas:

$$\begin{aligned} \mathbf{x}_v(t+1) &= f_{\mathbf{w}}(\mathbf{l}_v, \mathbf{l}_{co[v]}, \mathbf{x}_{ne[v]}, \mathbf{l}_{ne[v]}), \\ \mathbf{o}_v &= g_{\mathbf{w}}(\mathbf{x}_v(t), \mathbf{l}_v). \end{aligned} \tag{2.8}$$

The computation procedure outlined in Equation 2.8 can be visualized as an *encoding network*, as described in Scarselli's work [27]. In this network, each graph node is replaced by a pair of units responsible for calculating the functions $f_{\mathbf{w}}$ and $g_{\mathbf{w}}$ defined in Equation 2.5. For every node v, the first unit, upon activation, computes the state value $\mathbf{x}_v(t+1)$ based on the information stored by its neighboring nodes and the node itself (e.g., the previous state value $\mathbf{x}_v(t)$). On the other hand, the second unit is responsible for determining the output value $\mathbf{o}_v(t)$ of the node, considering the current state value $\mathbf{x}_v(t)$ and the label $\mathbf{l}_v$ associated with node v. The network structure is visually depicted in Figure 2.3. It is worth noting that if the functions $f_{\mathbf{w}}$ and $g_{\mathbf{w}}$ are implemented using a traditional feed-forward neural network, the encoding network becomes a recurrent neural network (RNN), where the connections between neurons can be categorized as internal or external connections. The internal connections arise from the unit's network architecture, while the external connections are defined by the edges of the graph.

2.4.3 Learning procedure

Before delving into the specific formulas and calculations used for training a RecGNN, a concise overview of the gradient calculation technique for RNNs is first shown. This technique is particularly relevant to the training of RecGNNs, due to the fact that if the units comprising the encoding network are implemented using traditional feed-forward networks, the encoding network itself can be classified as an RNN. Lastly, a comprehensive outline of the necessary steps needed for the training process is provided.

Recurrent neural networks and the backpropagation through time step

As discussed earlier, if the units within the encoding network are implemented using traditional feed-forward networks, the encoding network itself can be classified as a recurrent neural network. One key characteristic of this network is its ability to make predictions based not only on the current inputs but also on the historical inputs that preceded them. This is achieved through the presence of directed cycles within the neural connections of an RNN. However, this cyclic structure poses a

challenge when it comes to training the network using the standard backpropagation algorithm. To address this challenge, a technique called *Backpropagation through time* (BPTT) is employed, specifically designed to handle sequential data's temporal nature. In BPTT, the network is effectively 'unrolled' across different time steps, resulting in the creation of multiple network copies, each corresponding to a specific time step. This unrolling process transforms the cyclic connections into a directed acyclic graph (DAG). Consequently, during the training process with BPTT, the system is trained not only taking into account a particular time t but also taking into consideration all the preceding time steps, such as $t-1, t-2, t-3$, and so on. By considering the entire temporal context, the RNN can effectively learn and make predictions based on the sequential dependencies within the data.

Training a GNN

The objective of the training process is to find the optimal set of parameters $\mathbf{w}$ such that the function $\phi_{\mathbf{w}}$ closely approximates the data within the learning dataset defined in Equation 2.4. To this end, the work of Scarselli [27] proposes a standard quadratic cost function to be minimized in the parameters $\mathbf{w}$:

$$e_{\mathbf{w}} = \sum_{i=1}^{p} \sum_{j=1}^{N_i} (\mathbf{t}_{i,j} - \phi_{\mathbf{w}}(G_i, v_{i,j}))^2 \tag{2.9}$$

The term $(\mathbf{t}_{i,j} - \phi_{\mathbf{w}}(G_i, v_{i,j}))^2$ refers to the squared difference between the target value $(\mathbf{t}_{i,j})$ of a node v_i and the estimated embedding of the node. This difference is evaluated for every node $v_{i,j} \in V_i$ and every graph $G_i = (v_i, E_i)$ belonging to the training dataset.

The learning algorithm, which relies on the Backpropagation Through Time described before, consists of the following steps:

1. *Forward step*: by employing the equation given in Equation 2.7, the states $\mathbf{x}_v(t)$ undergo iterative updates until reaching a time T where $\mathbf{x}(T)$ closely approximates the fixed point solution of Equation 2.6, i.e., $\mathbf{x}(T) \approx \mathbf{x}$.

2. *Backpropagation step*: by defining an auxiliary variable $\mathbf{z}(t)$ and by making it converge to a value $\mathbf{z}$, the gradient of the error is evaluated: $\partial e_{\mathbf{w}}(T)/\partial \mathbf{w}$.

3. *Updating step*: By utilizing the previously obtained error gradient, update the weights $\mathbf{w}$ following a conventional gradient descent equation.

As said in the above list, the auxiliary variable $\mathbf{z}(t)$ is used to find the gradient of the error. Particularly, given that the global transition and output functions

$F_{\mathbf{w}}(\mathbf{x}, \mathbf{l}), G_{\mathbf{w}}(\mathbf{w}, \mathbf{l}_N)$ are differentiable, $\mathbf{z}(t)$ is defined as:

$$\mathbf{z}(t) = \mathbf{z}(t+1) \cdot \frac{\partial F_{\mathbf{w}}}{\partial \mathbf{x}}(\mathbf{x}, \mathbf{l}) + \frac{\partial e_{\mathbf{w}}}{\partial \mathbf{o}} \cdot \frac{\partial G_{\mathbf{w}}}{\partial \mathbf{x}}(\mathbf{x}, \mathbf{l}_N). \tag{2.10}$$

An interesting theorem, whose proof can be found in [27], states that the sequence $\mathbf{z}(T), \mathbf{z}(T-1), \mathbf{z}(T-2), \ldots$ converges exponentially fast to a value $\mathbf{z} = \lim_{t \to -\infty} \mathbf{z}(t)$, independently from the initial state $\mathbf{z}(T)$. Moreover, the following equation for calculating the gradient of the error holds:

$$\frac{\partial e_{\mathbf{w}}}{\partial \mathbf{w}} = \frac{\partial e_{\mathbf{w}}}{\partial \mathbf{o}} \cdot \frac{\partial G_{\mathbf{W}}}{\partial \mathbf{w}}(\mathbf{x}, \mathbf{l}_N) + \mathbf{z} \cdot \frac{\partial F_{\mathbf{W}}}{\partial \mathbf{w}}(\mathbf{x}, \mathbf{l}). \tag{2.11}$$

The contribution to the gradient from the output function $G_{\mathbf{w}}$ is represented by the first term on the right-hand side of Equation 2.11. Conversely, the second term represents the contribution from the transition function $F_{\mathbf{w}}$.

2.5 Convolutional Graph Neural Network

In addition to the analysis of Recurrent GNNs discussed earlier, another significant variant of GNNs is the Convolutional GNN. ConvGNNs adopt the fundamental principles of conventional *Convolutional Neural Networks* (CNNs), where feature extraction is accomplished using multiple filters. These filters traverse the data and execute the convolution operation. However, graph data is non-Euclidean as the distances between nodes do not adhere to the Euclidean space principles. The relationships between nodes are determined by arbitrary edges or connections, and the distances between nodes are not necessarily defined by straight lines or continuous geometric characteristics. Consequently, due to the non-Euclidean nature of graphs, the standard convolution operation performed by CNN filters cannot be directly applied. To address this, a novel convolution method called *graph convolution* is employed, which is realized using *convolutional filters*.

The classification of CNNs can be categorized into two main groups: spectral-based and spatial-based. The key distinction between these two types lies in their respective definitions of graph convolution. This section provides an examination and analysis of these two classifications.

As specified in Section 2.4, the notations employed in Section 2.4 and in the following sections are different. Indeed, Section 2.4, which deals with RecGNNs, employs the original notation of Scarselli [27] while the following sections, which deal with ConvGNNs, use the notations of [13].

2.5.1 Spectral-based ConvGNN

Standard convolutional filters, employed in CNNs, exhibit translation invariance, enabling weight sharing where the same filter is applied across different parts of the input [33]. This property allows them to identify similar features regardless of their specific spatial positions in the spatial domain. In contrast, graphs lack a well-defined spatial concept or a mathematical definition for spatial translation. *Spectral graph convolution*, denoted as $*_G$, helps in this direction since it provides a mathematical framework to design operators (filters) with the translation-invariant property [34].

Let $G(V, E)$ be a given graph, with its adjacency matrix denoted as $\mathbf{A}$ and its degree matrix as $\mathbf{D}$. The graph G can be represented using the normalized graph Laplacian matrix, defined as $\mathbf{L} = \mathbf{I}_N - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$. Since $\mathbf{L}$ is positive semidefinite, it possesses precisely N orthogonal eigenvectors that form a basis capable of diagonalizing the Laplacian matrix. Therefore, $\mathbf{L}$ can be decomposed as $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$, where $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \dots \mathbf{u}_{N-1}]$ represents a unitary matrix containing the eigenvalues of $\mathbf{L}$, and $\mathbf{\Lambda}$ is a diagonal matrix containing the corresponding eigenvalues.
In the context of graph signal processing, a graph signal $\mathbf{x} \in \mathbb{R}^N$ represents a feature vector associated with all the nodes of a graph. The graph Fourier transform of a signal $\mathbf{x}$ is defined as $\mathcal{F}(\mathbf{x}) = \mathbf{U}^T\mathbf{x}$, and the corresponding inverse operation is defined as $\mathcal{F}^{-1}(\hat{\mathbf{x}}) = \mathbf{U}\hat{\mathbf{x}}$. The graph Fourier transform maps a signal $\mathbf{x}$ into the orthonormal space defined by the eigenvectors of the Laplacian matrix.

Ultimately, the spectral graph convolution between a graph signal $\mathbf{x}$ and a filter $\mathbf{g}$ can now be defined as follows:

$$\mathbf{x} *_G \mathbf{g} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{g})) = \mathbf{U}(\mathbf{U}^T\mathbf{x} \odot \mathbf{U}^T\mathbf{g}), \tag{2.12}$$

where $\odot$ denotes the element-wise Hadamard product. Introducing the filter as $\mathbf{g}_{\boldsymbol{\theta}} = diag(\mathbf{U}^T\mathbf{g})$, Equation 2.12 simplifies to:

$$\mathbf{x} *_G \mathbf{g}_{\boldsymbol{\theta}} = \mathbf{U}\mathbf{g}_{\boldsymbol{\theta}}\mathbf{U}^T\mathbf{x}. \tag{2.13}$$

The aforementioned equation serves as the foundation for any Spectral-based ConvGNN, with different methodologies diverging in their selection of the filter $\mathbf{g}_{\boldsymbol{\theta}}$.

Spectral CNN

In the study of [35], which introduced the initial Spectral-based ConvGNN known as the *Spectral CNN*, the filter is defined as a diagonal matrix comprising trainable parameters $\mathbf{\Theta}_{i,j}^{(k)}$. These parameters within $\mathbf{\Theta}_{i,j}^{(k)}$ correspond to layer k and are

responsible for mapping the ith input feature to the jth output feature. In layer k, the resulting output of a hidden layer is expressed as:

$$\mathbf{H}_{:,j}^{(k)} = \sigma(\sum_{i=1}^{c_{k-1}} \mathbf{U}\mathbf{\Theta}_{i,j}^{(k)}\mathbf{U}^T\mathbf{H}_{:,i}^{(k-1)}), \qquad j = 1,2,\ldots,c_k, \tag{2.14}$$

where c_{k-1} and c_k indicate the number of input and output channels respectively. Unfortunately, Spectral CNNs suffer from a significant computational complexity due to the eigendecomposition of the Laplacian matrix, which has a computational cost of $\mathcal{O}(N^3)$. Furthermore, any modification in the graph structure necessitates recalculating the eigenvectors and eigenvalues of the Laplacian matrix $\mathbf{L}$, further amplifying the computational overhead.

Chebyshev Spectral CNN

In the context of Spectral CNN, in addition to the limitations mentioned earlier, the filters used are generally not localized. As a result, these filters do not confine their influence solely to the closest K neighbors of a node but instead involve all nodes in the graph. However, this drawback is addressed by Chebyshev Spectral CNN (ChebNet) [36], which leverages Chebyshev polynomials. These polynomials are recursively defined as $T_i(\mathbf{x}) = 2\mathbf{x}T_{i-1}(\mathbf{x}) - T_{i-2}(\mathbf{x})$ with initial values $T_0(\mathbf{x}) = 1$ and $T_1(\mathbf{x})$. They are employed to approximate the filter $\mathbf{g}_{\boldsymbol{\theta}}$ as follows:

$$\mathbf{g}_{\boldsymbol{\theta}} = \sum_{i=0}^{K-1} \boldsymbol{\theta}_i T_i(\tilde{\mathbf{\Lambda}}), \quad \tilde{\mathbf{\Lambda}} = 2\mathbf{\Lambda}/\lambda_{\max} - \mathbf{I}_N. \tag{2.15}$$

By introducing a matrix $\tilde{\mathbf{L}} = 2\mathbf{L}/\lambda_{\max} - \mathbf{I_N}$, where $T_i(\tilde{\mathbf{L}}) = \mathbf{U}T_i(\tilde{\mathbf{\Lambda}})\mathbf{U}^T$, Equation 2.13 can be rewritten as:

$$\mathbf{x} *_G \mathbf{g}_{\boldsymbol{\theta}} = \sum_{i=0}^{K-1} \boldsymbol{\theta}_i T_i(\tilde{\mathbf{\Lambda}})\mathbf{x}. \tag{2.16}$$

It's important to note that in Equation 2.16, the sum is truncated to a value $K \ll N$, resulting in complexity that is independent of the graph size. Furthermore, this truncation preserves the localization property.

Graph Convolutional Network

The main drawback of ChebNet lies in the large number of parameters $\boldsymbol{\theta}$ that are employed. Thus, complexity still remains an issue and the model may run into overfitting scenarios. *Graph Convolutional Network* (GCN) [37] addresses this problem by introducing a first-order approximation of ChebNet (i.e. $K = 1$) and assuming $\boldsymbol{\theta} = \boldsymbol{\theta}_0 = -\boldsymbol{\theta}_1$. These modifications lead to the following simplification of Equation 2.16:

$$\mathbf{x} *_G \mathbf{g}_{\boldsymbol{\theta}} = \boldsymbol{\theta}(\mathbf{I}_N + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}). \tag{2.17}$$

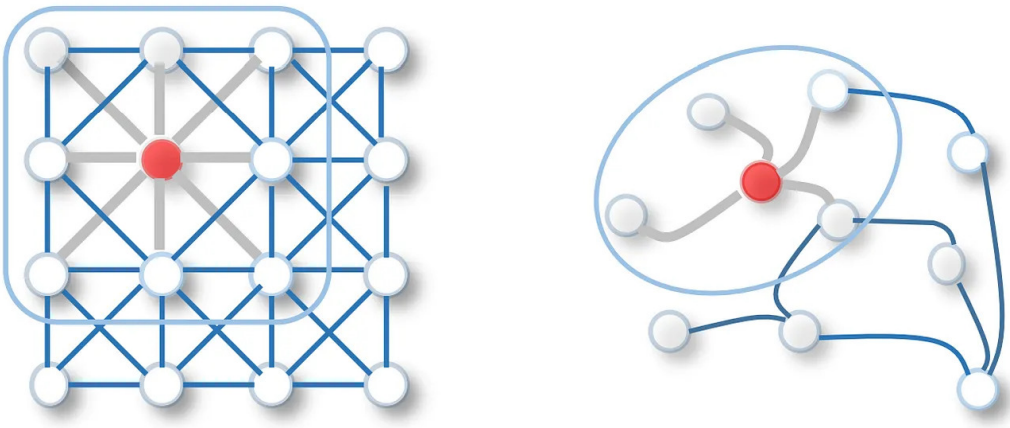

Figure 2.4: The left image shows a standard convolution performed by a CNN. A 3×3 filter is centered in one pixel (or node) and computes the weighted average of the central pixel and its 8 neighbors. The right image shows a spatial-based graph convolution, where the filter is centered in one node and computes the weighted average of the node and its neighbors. In the right image, the size of the filter varies with the number of neighbors.

2.5.2 Spatial-based ConvGNN

Spatial-based ConvGNNs shares similarities with standard CNNs as both involve convolutional operations defined based on the spatial relationships between nodes. In CNNs, each pixel in an image can be viewed as a node in a graph, connected to its eight adjacent pixels, as illustrated in the left image of Figure 2.4. Subsequently, a 3×3 filter is applied to a node (or pixel), followed by a weighted average of the node and its eight neighbors. Similarly, in spatial-based graph convolution, a weighted average of a node and its neighbors is computed, as shown in the right image of Figure 2.4. However, unlike the fixed-size filters used in CNNs, the spatial-based graph convolution does not have a predetermined filter size due to the varying number of neighbors that nodes can have in a graph.

Neural Network for Graphs

The initial proposal for spatial-based ConvGNN is the *Neural Network for Graphs* (NN4G) [38]. In contrast to Recurrent GNN, NN4G adopts a feedforward architecture, eliminating any recursive or feedback connections. Moreover, NN4G takes a constructive approach, where node information is incrementally utilized without introducing cyclic dependencies in the definition of system state variables, as opposed to Equation 2.6. Consequently, NN4G employs the following equation

to compute the states of the next layer of nodes:

$$\mathbf{h}_v^{(k)} = \text{act}(\mathbf{W}^{(k)^T}\mathbf{x}_v + \sum_{i=1}^{k-1}\sum_{u\in\mathcal{N}(v)} \mathbf{\Theta}^{(k)^T}\mathbf{h}_u^{(k-1)}), \tag{2.18}$$

where act is an activation function, $\mathbf{W}^{(k)^T}, \mathbf{\Theta}^{(k)^T}$ are learnable parameters, and $\mathbf{h}_v^{(k)}$ is the output of the hidden layer for v and layer k. Unfortunately, NN4G works directly with the adjacency matrix without normalizing it, with the consequence that node states may assume extremely different values.

2.5.3 Diffusion Convolutional Neural Network

The Diffusion Convolutional Neural Network (DCNN) [39] interprets graph convolution as a diffusion process, where the exchange of information between nodes aims to achieve equilibrium through multiple information exchange steps. The exchange of information is governed by transition probabilities, which are gathered into the probability transition matrix $\mathbf{P}$ defined as $\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$. Ultimately, DCNN defines diffusion graph convolution as follows:

$$\mathbf{H}^{(k)} = \text{act}(\mathbf{W}^{(k)} \odot \mathbf{P}^k\mathbf{X}). \tag{2.19}$$

Next, the obtained $\mathbf{H}^{(1)}, \mathbf{H}^{(2)}, \ldots, \mathbf{H}^{(K)}$ are combined through concatenation to produce the final outputs of the model.
However, the primary limitation of DCNN originates from the transition matrix, which diminishes the contribution of peripheral nodes in comparison to central nodes.

Message Passing Neural Networks

When considering *Message Passing Neural Networks* (MPNN) [40], graph convolution is viewed as a message-passing procedure, allowing edges to transmit information directly between nodes. To extend the propagation of information, the message-passing process is iterated for K steps. Consequently, the following equation is employed to determine the hidden layer output of a node v:

$$\mathbf{h}_v^{(k)} = U_k(\mathbf{h}_v^{k-1}, \sum_{u\in\mathcal{N}(v)} M_k(\mathbf{h}_v^{(k-1)}, \mathbf{h}_u^{(k-1)}, \mathbf{x}_{vu}^e)), \tag{2.20}$$

where U_k and M_k represent functions that contain learnable parameters. After obtaining the hidden representation for each node v, the resulting value $\mathbf{h}_v^{(K)}$ can be utilized in two ways: either passed to an output layer for calculating the final node embedding or employed in a readout function to determine the ultimate graph embedding.

Chapter 3

Proposed algorithm

The objective of this work is to discover a new approach for addressing two fundamental graph theory problems: the Maximum Independent Set and the Minimum Vertex Cover. To achieve this, two algorithms, whose fundamental steps are described in Section 3.1, inspired by DP techniques have been developed. Moreover, the algorithms leverage the capabilities of a Machine Learning model that acts as a *comparator*. If the comparator, whose operating principles are described in Section 3.2, is properly trained, the algorithms can lead to near-optimal solutions with a low computational complexity.

3.1 Algorithm steps

The algorithms for solving the MIS and the MVC problems are built on fundamental operating principles, like the knowledge of sub-graph or the branching and ending steps. Sections 3.1.1 and 3.1.2 describe the steps of the algorithms for, respectively, the MIS and MVC problems.

3.1.1 Algorithmic operations for MIS

As previously mentioned, the algorithm for MIS effectively employs DP techniques. Consequently, it recursively decomposes the original graph $G(V, E)$ into a collection of progressively smaller sub-graphs, namely $G_1(V_1, E_1), \ldots, G_i(V_i, E_i), \ldots$. Then, the algorithm stops and identifies a solution once a sub-graph takes on a specific structure. Thus, the fundamental essence of the algorithm hinges upon this decomposition, or branching, process, which necessitates these sub-graphs to be

chosen properly. The branching process, or *branching step*, and the meaning of choosing a sub-graph "properly" are summarized in the following theorem:

Theorem 1. *Let a graph $G(V, E) \in \mathcal{G}$. Then for any vertex $v \in V$ with $d(v) \geq 1$,*

$$|\text{MIS}(G)| = \max\left(|\text{MIS}\left(G/\mathcal{N}(v)\right)|, |\text{MIS}(G/\{v\})\,|\right)^{1}.$$

Proof. Let $G(V, E)$ be a graph and $\text{MIS}(G)$ be its maximum independent set. The goal is to show that for any vertex $v \in V$ with $d(v) \geq 1$, the size of $\text{MIS}(G)$ can be obtained by either removing v or removing its neighbors $\mathcal{N}(v)$.

Consider two cases:

- $v \in \text{MIS}(G)$

  In this case, if $\mathcal{N}(v)$ is removed from G, the resulting graph is denoted as $G_0 = G \setminus \mathcal{N}(v)$. Since the neighbors of v cannot be in the maximum independent set, removing it does not affect the size of $\text{MIS}(G)$. Therefore, $\text{MIS}(G) = \text{MIS}(G_0)$.

- $v \notin \text{MIS}(G)$

  In this case, v is removed from G to obtain the graph $G_1 = G \setminus \{v\}$. Since v is not in the maximum independent set, removing it does not affect the size of $\text{MIS}(G)$. Therefore, $\text{MIS}(G) = \text{MIS}(G_1)$.

By considering these two cases, it is shown that for any vertex $v \in V$, the maximum independent set $\text{MIS}(G)$ can be obtained by either removing v or removing its neighbors $\mathcal{N}(v)$. Thus, the size of the maximum independent set can be expressed as follows:

$$|\text{MIS}(G)| = \max\left(|\text{MIS}\left(G/\mathcal{N}(v)\right)|, |\text{MIS}(G/\{v\})\,|\right).$$

□

According to Theorem 1, the branching step mentioned above consists of picking a random node v from a sub-graph $G(V, E)$ and building two graphs G_0 and G_1. G_0 is the original graph G without the neighbors of v, namely $G_0 = G/\mathcal{N}(v)$, while

[1] Note: In the trivial case where G is an empty graph (i.e., it has no edges), the size of the maximum independent set is $|V|$.

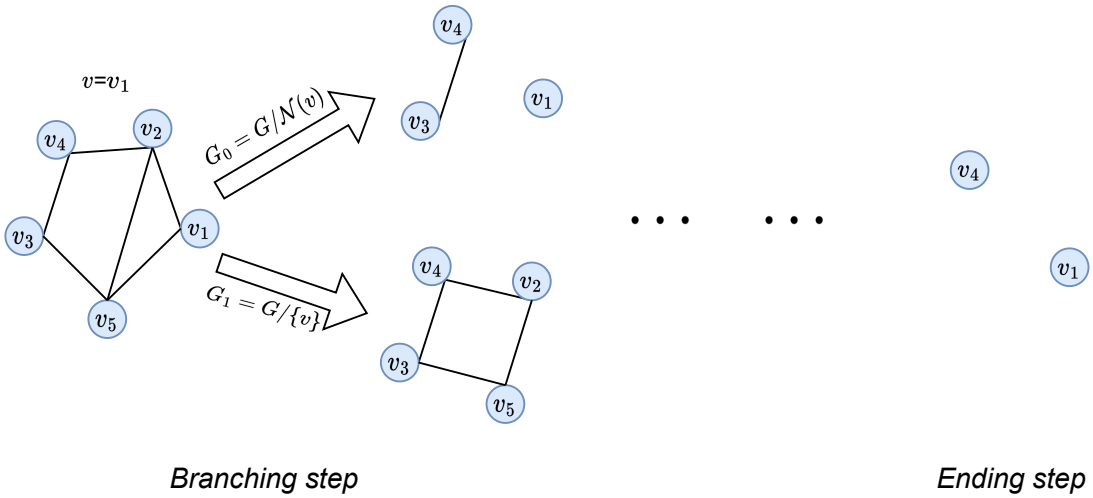

Figure 3.1: The figure shows an example of branching and ending steps in the case of the algorithm for MIS and a specific graph. The left part of the figure shows the branching step if the randomly chosen node is v_1. The right part shows the ending step, as the sub-graph is composed of two isolated nodes.

G_1 is the original graph G without the node v, namely $G_1 = G/\{v\}$. Then, the MIS of G corresponds to the MIS of G_0 if $|\text{MIS}(G_0)| \geq |\text{MIS}(G_1)|$ or to the MIS of G_1 if $|\text{MIS}(G_0)| < |\text{MIS}(G_1)|$.

The aforementioned branching step is recursively applied by the algorithm for MIS. For each step i of the algorithm, from a sub-graph G two graphs G_0 and G_1 are built in the manner specified above. Then, the algorithm for MIS tries to predict if $|\text{MIS}(G_0)| \geq |\text{MIS}(G_1)|$ is true or not. If true, G_0 is the sub-graph, otherwise G_1 it is. Once the sub-graph is found, it is used in step $i+1$ of the algorithm to perform a new decomposition. Whenever a sub-graph has no edges, all nodes are isolated and therefore independent. This last step, denoted as *ending step*, makes the algorithm stop and the solution found by the algorithm coincides with the total number of nodes in the graph.

The left part of Figure 3.1 shows the branching step of a specific sub-graph in case the node $v = v_1$ is randomly picked. Graph G_0 is composed of the nodes v_1, v_3, and v_4 as the neighbors v_2, v_5 of v are removed. Graph G_1, instead, does not include v_1, as it is removed from the original sub-graph G. The right part of Figure 3.1 depicts the ending step of the algorithm. The sub-graph has two isolated nodes and, thus, the algorithm stops and the solution given by the algorithm is equal to the total number of isolated nodes, that is 2 in the specific example of Figure 3.1.

3.1.2 Algorithmic operations for MVC

The algorithmic approach for solving the MVC problem shares similarities with the algorithm employed for addressing the MIS problem. Specifically, the algorithm for MVC strives to diminish the inherent complexity of the initial graph by iteratively identifying optimal sub-graphs. However, the two graphs G_0 and G_1, formed during the branching step, do not undergo the same computational treatment as elaborated in Section 3.1.1. Instead, the notion of a "copy node" is introduced. A copy node v' represents a synthetic node that is introduced into the graph to reduce the original graph's structure in terms of edge count. The synthetic node v' is introduced whenever a node v has been identified as part of the final solution. In this context, all edges incident on v are removed, and the copy node v' is inserted in the graph as well as the edge (v, v').

The following theorem, whose proof is omitted as it is similar to the one of Theorem 1, describes how the graphs G_0 and G_1 are built and the operating principle behind the branching step:

Theorem 2. *Let a graph $G(V, E) \in \mathcal{G}$. Then for any vertex $v \in V$ with $d(v) \geq 1$,*

$$|\mathrm{MVC}(G)| = \min\left(|\mathrm{MVC}\left(G_0\right)|, |\mathrm{MVC}(G_1\})\,|\right)^{2}.$$

where

- *$G' := G(V \setminus \{v\}, E \setminus \{(u, \ell) \mid u \in \mathcal{N}(v) \wedge \ell \in \mathcal{N}(u)\})$, G' is created by removing v from G as well as all the edges incident to neighbors of v.*
- *$G_0 := G'(V \cup \{u' | u \in \mathcal{N}(v)\}, E \cup \{(u', u) | u \in \mathcal{N}(v)\})$, G_0 is created from G' by adding the copy nodes $u' | \forall u \in \mathcal{N}(v)$ and the edges (u', u) for all $u \in \mathcal{N}(v)$ (each vertex u is connected with its copy u'). G_0 represents the situation where vertex v is not selected in the MVC, but its neighbors are.*
- *$G_1 = G(V \cup \{v'\}, E \setminus \{(u, v) | u \in \mathcal{N}(v)\} \cup \{(v', v)\})$, G_1 is created from G by removing all edges incident to v and adding a copy node v' that is then connected to v. G_1 represents the situation where v is selected to be part of the MVC.*

The *branching step* of the algorithm for MVC consists of analyzing a sub-graph G. From G, two graphs G_0 and G_1 are built according to Theorem 2 and the

[2] In the trivial case where G is a graph with only connected components of two vertices ($d(v) \leq 1$ for all $v \in V$), the size of the minimum vertex cover is $|G| = |E(G)|$.

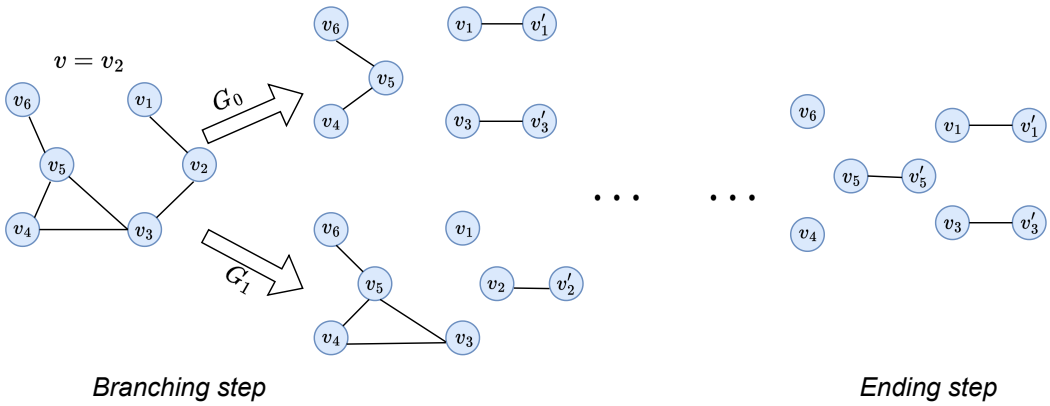

Figure 3.2: The figure depicts the branching and ending steps for the algorithm for MVC and for a specific example. The left part shows the generation of the graph G_0 and G_1 if a random node $v = v_2$ is picked. The right part shows the ending step, where the sub-graph has edges with degrees less or equal to 1.

algorithm for MVC aims at estimating which graphs, among G_0 and G_1, has the lowest MVC. The algorithm progression stops with the *ending step*, which occurs whenever all nodes of a sub-graph have a degree lower or equal to one. At this point, the solution found by the algorithm coincides with the total number of edges of the sub-graph.

Figure 3.2 depicts the previously mentioned branching and ending steps for a specific example. Specifically, a node $v = v_2$ is picked. The graph G_0 is generated by removing the node v_2 and by removing all edges that incident on the neighbors of v_2, that are nodes v_1, v_3. Then, the copy nodes v_1', v_3' are inserted and connected to v_1, v_3 respectively. Graph G_1 is obtained by removing the edges incident on node v_2 and by adding a copy node v_2' connected to v_2. The sub-graph of the right part of Figure 3.2 has nodes of degrees less or equal to 1, thus the algorithm stops (ending step) and the solution found by the algorithm is 3 since 3 edges are present in the sub-graph.

3.2 Graph-comparing function and comparator function

The biggest challenge of the algorithms described in the previous sections is posed by the branching step. This procedure tries to estimate which graph between G_0 and G_1 is the sub-graph that is used in the next iteration of the algorithm. The following sections describe the tool that is used to perform such an estimate and

make a detailed overview of the algorithms for MIS and MVC.

3.2.1 Graph comparing function (and comparator) for MIS

Consider a function, denoted as *Graph-Comparing Function for MIS*, that takes two graphs as input and outputs either a 0 or a 1:

$$\mathrm{CMP}^{\mathrm{MIS}} : \mathcal{G} \times \mathcal{G} \longmapsto \{0,1\}. \tag{3.1}$$

The graph-comparing $\mathrm{CMP}^{\mathrm{MIS}}$ function defined above compares two graphs G_0 and G_1 based on the size of their MIS. Namely, if $|\mathrm{MIS}(G_0)| \geq |\mathrm{MIS}(G_1)|$ then $\mathrm{CMP}^{\mathrm{MIS}}(G_0, G_1) = 0$ while if $|\mathrm{MIS}(G_0)| < |\mathrm{MIS}(G_1)|$ then $\mathrm{CMP}(G_0, G_1) = 1$.

The function 3.1 would solve all problems, as it would always find the proper sub-graph with the largest MIS and, according to Theorem 1, it would always find the optimal MIS solution. Indeed, starting from a graph $G(V, E)$ and a random node $v \in V$, by recursively selecting either $G/\{v\}$ or $G/\mathcal{N}(v)$ based on $|\mathrm{MIS}(G/\{v\})| \geq |\mathrm{MIS}(G/\mathcal{N}(v))|$, the algorithm surely ends in an independent set of maximum size. The decision of whether $|\mathrm{MIS}(G/\{v\})| \geq |\mathrm{MIS}(G/\mathcal{N}(v))|$ at each recursive call can be made according to the output of $\mathrm{CMP}^{\mathrm{MIS}}(G/\{v\}, G/\mathcal{N}(v))$. Moreover, the following remarks hold:

Remark 1. *Given a graph-comparing function for MIS* $\mathrm{CMP}^{\mathrm{MIS}} : \mathcal{G} \times \mathcal{G} \mapsto \{0,1\}$, *the induced algorithm is randomized, since at Step* 4 *of Algorithm 1, a vertex* v *is randomly selected. Notice that Algorithm 1 recursively proceeds until a sub-graph with* 0 *edges is reached (see Step* 2*).*

Remark 2. *Two different graph-comparing functions* $\mathrm{CMP}^{\mathrm{MIS}}$ *and* $(\mathrm{CMP}^{\mathrm{MIS}})'$ *induce two different optimal algorithms* $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}}$ *and* $\mathcal{A}^{(\mathrm{CMP}^{\mathrm{MIS}})'}$ *for calculating the maximum independent set.*

The cornerstone of this work is that, as highlighted by Remark 1, any graph-comparing function $\mathrm{CMP}^{\mathrm{MIS}}$ induces an algorithm $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}}$ for MIS that behaves as displayed in Algorithm 1. Recursively selecting $G/\{v\}$ or $G/\mathcal{N}(v)$ based on the output of a graph comparing function $\mathrm{CMP}^{\mathrm{MIS}}(G/\{v\}, G/\mathcal{N}(v)) \in \{0, 1\}$ always guarantees to reach an independent set of a maximum size of the original graph. Unfortunately, implementing a graph-comparing function that behaves as described above is impossible. Nonetheless, a new function that behaves almost as a graph-comparing function can be designed, as described in the next paragraph.

In case $\mathrm{CMP}^{\mathrm{MIS}}(G_0, G_1) \neq \mathbb{I}\left[|\mathrm{MIS}(G_0)| < |\mathrm{MIS}(G_1)|\right]$, where $\mathbb{I}$ is the indicator function, it is not guaranteed that the induced algorithm produces an independent set

Algorithm 1 Induced Algorithm for the MIS problem

function $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}}(G(V,E))$ ▷ Algorithm $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}}(G)$ takes a graph G as input
  if $|E| = 0$ **then return** V
  end if
  pick a vertex $v \in V$ with $d(v) > 0$ uniformly at random.
  $G_0 \leftarrow G \setminus \{v\}$ and $G_1 \leftarrow G \setminus \mathcal{N}(v)$
  if $\mathrm{CMP}^{\mathrm{MIS}}(G_0, G_1) = 0$ **then**
    $G \leftarrow G_0$ ▷ Remove vertex v
  else
    $G \leftarrow G_1$ ▷ Remove the neighbors of v
  end if
  return $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}}(G)$
end function

of maximum size. In this scenario, the graph-comparing function for MIS is referred to as the *Comparator Function for MIS*, and the function is denoted as $\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}$. Naturally, the algorithm $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}}$, described in 1, is still valid even if a comparator function, not a graph-comparing function, is employed. Simply, $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}}$ does not guarantee to produce independent sets of maximum size, differently from $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}}$. Moreover, Remarks 1,2 still apply to the comparator function $\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}$, as pointed out by the following Remark:

Remark 3. *A graph-comparing function for MIS, denoted as* $\mathrm{CMP}^{\mathrm{MIS}}$*, always outputs the result of* $\mathbb{I}\left[|\mathrm{MIS}(G_0)| < |\mathrm{MIS}(G_1)|\right]$*. A comparator function for MIS, denoted as* $\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}$*, does not guarantee to output* $\mathbb{I}\left[|\mathrm{MIS}(G_0)| < |\mathrm{MIS}(G_1)|\right]$*. Nevertheless, Remarks 1 and 2 and the Algorithm 1 are still valid for the comparator function.*

As mentioned above, the difference between $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}}$ and $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}}$ is that the first one will always lead to an independent set that is maximum, while the second one does not guarantee to lead to an independent set that is maximum.

Even if the algorithm induced by the comparator does not guarantee to lead to an optimal solution, the following considerations still apply:

- the comparator for MIS $\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}$ can be efficiently designed in order to have low computational complexity.
- the comparator for MIS $\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}$, if properly designed, can induce an algorithm

for MIS $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}}$ that leads to near-optimal solutions.

At this point, the question that arises is how to design a comparator function for MIS that meets the two points above and resembles as much as possible a graph-comparing function. Surprisingly, this work employs a Machine Learning model $M_{\boldsymbol{\theta}} : \mathcal{G} \mapsto \mathbb{R}$ to effectively design a comparator:

$$\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}(G_0, G_1) = \mathbb{I}[M_{\boldsymbol{\theta}}(G_0) < M_{\boldsymbol{\theta}}(G_1)]. \tag{3.2}$$

Since the model is parameterized by a set of parameters $\boldsymbol{\theta}$, the resulting comparator is characterized by the same symbol: $\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}$.

The structure of the model $M_{\boldsymbol{\theta}}$, used for designing the comparator $\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}$, is analyzed in Chapter 4.

3.2.2 Graph-comparing function (and comparator) for MVC

Similarly to what was discussed in Section 3.2.1, the *Graph-Comparing Function for MVC* is the knowledge behind the algorithm for solving the MVC problem. However, if on one side the function $\mathrm{CMP}^{\mathrm{MVC}}$ for MVC is defined in the same way as Equation 3.1, on the other side its use is different. Indeed, the graph-comparing function $\mathrm{CMP}^{\mathrm{MVC}}$ compares two graphs G_0 and G_1 based on the size of their MVC. If $|\mathrm{MVC}(G_0)| \leq |\mathrm{MVC}(G_1)|$ then the graph-comparing function assumes a value equal to zero, namely $\mathrm{CMP}^{\mathrm{MVC}}(G_0, G_1) = 0$. Vice versa, if $|\mathrm{MVC}(G_0)| > |\mathrm{MVC}(G_1)|$ the graph-comparing function assumes a value equal to one, namely $\mathrm{CMP}^{\mathrm{MVC}}(G_0, G_1) = 1$.

As for Algorithm 1, a graph-comparing function for MVC induces a recursive algorithm $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}$, as depicted in Algorithm 2. For each iteration of the algorithm, a random node v is picked from a graph G. Then, two graphs G_0 and G_1 are built from G as explained by Theorem 2. According to the MVC of G_0 and G_1, the graph-comparing function $\mathrm{CMP}^{\mathrm{MVC}}$ chooses which of the two is a sub-graph. The new sub-graph becomes the graph G that is used in the next iteration.
The aforementioned process ends whenever a graph G is composed of either isolated nodes or nodes with a degree equal to one and, according to Theorem 2, a vertex cover of minimum size is attained.

Similarly to what is discussed in Section 3.2.1, the following remarks hold for the induced algorithm $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}$:

Remark 4. *Given a graph-comparing function for MVC* $\mathrm{CMP}^{\mathrm{MVC}} : \mathcal{G} \times \mathcal{G} \mapsto \{0,1\}$*, the induced recursive algorithm* $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}$ *is randomized, since at Step* 9 *of Algorithm*

2, a vertex v is randomly selected. Notice that Algorithm 2 recursively proceeds until a sub-graph with nodes having degree $d(v)$ lower than 1 is reached (see line 2 of 2).

Remark 5. *Two different graph-comparing functions for MVC* $\mathrm{CMP}^{\mathrm{MVC}}$ *and* $(\mathrm{CMP}^{\mathrm{MVC}})'$ *induce two different algorithms* $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}$ *and* $(\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}})'$ *for calculating the minimum vertex cover.*

Algorithm 2 Induced Algorithm for the MVC problem

1: **function** $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}(G(V,E))$ ▷ Algorithm $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}(G)$ takes a graph G as input
2: **if** $\forall v \in V : d(v) \leq 1$ **then** ▷ G is composed of isolated nodes and isolated edges
3: $S \leftarrow \varnothing$
4: **for** each edge $(v, v') \in E$ **do**
5: $S \leftarrow S \cup \{v\}$ ▷ Select one of the endpoints of each isolated edge
6: **end for**
7: **return** S
8: **end if**
9: pick a vertex $v \in V$ with $d(v) \geq 1$ uniformly at random.
10: $G' := G(V \setminus \{v\}, E \setminus \{(u, \ell) \mid u \in \mathcal{N}(v) \wedge \ell \in \mathcal{N}(u)\})$
11: $G_0 := G'(V \cup \{u' | u \in \mathcal{N}(v)\}, E \cup \{(u', u) | u \in \mathcal{N}(v)\}$
12: $G_1 = G(V \cup \{v'\}, E \setminus \{(u, v) | u \in \mathcal{N}(v)\} \cup \{(v', v)\})$
13: **if** $\mathrm{CMP}^{\mathrm{MVC}}(G_0, G_1) = 0$ **then**
14: $G \leftarrow G_0$ ▷ Remove vertex v and put it's neighbors in the MVC
15: **else**
16: $G \leftarrow G_1$ ▷ Put vertex v in the MVC and remove the edges to its neighbors
17: **end if**
18: **return** $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}(G)$
19: **end function**

Even for the MVC problem, a graph-comparing function cannot be implemented in practice, but it is possible to design an approximation of it. A *Comparator Function for MVC*, denoted as $\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}$, does not guarantee to always output the result of the indicator function $\mathbb{I}[|\mathrm{MVC}(G_0)| > |\mathrm{MVC}(G_1)|]$. Nevertheless, it is possible to design a comparator for MVC that induces an algorithm $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}}$ whose solutions are near-optimal and calculated in an efficient way. Moreover, $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}}$ shares the same properties of $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}}$ illustrated in Remarks 4 and 5:

Remark 6. *A graph-comparing function for MVC, denoted as* $\mathrm{CMP}^{\mathrm{MVC}}$*, always*

outputs the result of $\mathbb{I}\left[|\mathrm{MVC}(G_0)| > |\mathrm{MVC}(G_1)|\right]$. *A comparator function for MVC, denoted as* $\mathrm{CMP}_{\boldsymbol{\theta}}^{\mathrm{MVC}}$*, does not guarantee to output* $\mathbb{I}[|\mathrm{MVC}(G_0)| > |\mathrm{MVC}(G_1)|]$. *Nevertheless, Remarks 4 and 5 and the Algorithm 2 are still valid for the comparator function.*

Identically to the MIS problem, the comparator $\mathrm{CMP}_{\boldsymbol{\theta}}^{\mathrm{MVC}}$ is designed employing a Machine Learning model $M_{\boldsymbol{\theta}} : \mathcal{G} \mapsto \mathbb{R}$. Consequently, the resulting comparator assumes the following structure:

$$\mathrm{CMP}_{\boldsymbol{\theta}}^{\mathrm{MVC}}(G_0, G_1) = \mathbb{I}[M_{\boldsymbol{\theta}}(G_0) > M_{\boldsymbol{\theta}}(G_1)]. \tag{3.3}$$

The architecture of the ML model $M_{\boldsymbol{\theta}}$ for MVC is the same as the architecture of the model for MIS, and such architecture is elucidated in Chapter 4.

Chapter 4

Model architecture

As mentioned in Chapter 3, the algorithm employed for solving the MIS and MVC problems leverages an ML model. This model, denoted as $M_{\boldsymbol{\theta}}(G)$, is divided into two modules: the first module, denoted as *Graph Embedding Module* (GEM), contains GNNs while the second module is made of a standard *Fully Connected Neural Network* (FCNN). Section 4.1 describes the GEM module, while Section 4.3 provides an overview of the entire model architecture. Moreover, Section 4.2 illustrates an analysis of the GELU function, which is a peculiar activation function employed for the GEM module.

The next sections do not include a diversification between the model for MIS and for MVC, since both models share the same architecture. Consequently, in this chapter, every symbolic expression refers to both the MIS and MVC problems.

4.1 Graph Embedding Module (GEM)

The operating principle of the GEM is similar with respect to standards GNNs introduced in Section 2.5. Both GEM and GNNs consider a graph as input and output a representation of the graph, denoted as embedding. Nonetheless, GEM and standard GNNs differ in the way the embedding is computed. Indeed, unlike classic GNN modules, the GEM module captures different semantic meanings by leveraging three (not two as in standard GNNs) diverse types of information: the information contained in a node, its neighbors, and its anti-neighbors. This can be noted in the following recursive formula, which is at the heart of the GEM

functioning:

$$\boldsymbol{\mu}_v^{k+1} = \mathrm{LN}\left(\mathrm{GELU}\left(\left[\boldsymbol{\theta}_0^k\boldsymbol{\mu}_v^k \middle\| \boldsymbol{\theta}_1^k \sum_{u\in\mathcal{N}(v)} \boldsymbol{\mu}_u^k \middle\| \boldsymbol{\theta}_2^k \sum_{u\notin\mathcal{N}(v)} \boldsymbol{\mu}_u^k\right]\right)\right) . \tag{4.1}$$

In the above equation, $\boldsymbol{\mu}_v^{k+1} \in \mathbb{R}^{3p}$ denotes the embedding at iteration $k+1$ of the graph node v. $\boldsymbol{\theta}_0^k, \boldsymbol{\theta}_1^k, \boldsymbol{\theta}_1^k \in \mathbb{R}^{p\times 3p}$ are the parameters of the linear k-linear layers, LN is a normalization layer and GELU is particular activation function whose functioning is described in Section 4.2. The bias term is omitted in the equation for readability purposes.
Initially, all nodes in the graph have zero embeddings $\boldsymbol{\mu}_v^0 = \vec{0}$. Here, $\boldsymbol{\mu}_v^0 = \vec{0}$ denotes the initial embedding vector of node v. Then, for all iterations $k \in [0, \ldots, K-1]$ Equation 4.1 is computed in order to compute the final embedding $\boldsymbol{\mu}_v = \boldsymbol{\mu}_v^K$ for every node v of the graph. For every iteration k of the GEM module and for every node v, three different quantities are computed by three different linear layers. The first linear layer captures the embedding of node v, namely $\boldsymbol{\theta}_0^k\boldsymbol{\mu}_v^k$. The second linear layer computes the relationships between the embeddings of the neighbors of v by calculating the quantity $\boldsymbol{\theta}_1^k \sum_{u\in\mathcal{N}(v)} \boldsymbol{\mu}_u^k$. Finally, thanks to the third linear layer, the information on the anti-neighbors is also taken into account by the term $\boldsymbol{\theta}_2^k \sum_{u\notin\mathcal{N}(v)} \boldsymbol{\mu}_u^k$. The three linear layers of an iteration k are gathered into a bigger layer denoted as *Macro-Layer* and indicated with the symbol MacL_k.
The use of separate linear layers for different features (nodes, neighbors, and anti-neighbors embeddings), emphasizes the contrasting semantic meaning between neighbors and anti-neighbors, representing positive and negative relationships in the graph. Then, the results of the three linear layers are concatenated ($[\ldots\|\ldots]$ symbol in Equation 4.1) and passed to a GELU [41] activation function. A normalization layer prevents the embeddings from assuming too big or too small values.

After K iterations of Equation 4.1, every node of the graph has an embedding $\boldsymbol{\mu}_v$. In order to find the final graph embedding $\boldsymbol{\mu}_G$, an average pooling is performed according to the following equation:

$$\boldsymbol{\mu}_G = \frac{1}{|V|}\sum_{i\in V} \boldsymbol{\mu}_i. \tag{4.2}$$

At this point, the final graph embedding $\boldsymbol{\mu}_G$ is computed and can be passed to the FCNN module.

4.2 GELU activation function

Equation 4.1 employs a non-canonical activation function, namely the *Gaussian Error Linear Unit* (GELU). The GELU activation function has been proposed by

[41] and aims to combine properties from ReLUs, dropout, and zoneout.

ReLU activation function multiplies an input x by either 0 or 1, depending on the sign of x. It is important to remark that such multiplication is made deterministically, as the multiplication is only related to $x \gtrless 0$. Differently from ReLU, the zoneout [42] regularizer performs the multiplication only by 1 and does it stochastically. Also, by turning off neurons, the dropout technique performs the multiplication of an input x stochastically, but the multiplied value is 0.

The features of ReLUs, dropout, and zoneout are merged by GELU by stochastically multiplying the input by zero or one, but the output value is obtained deterministically. The neuron input x is multiplied by a value m obtained according to the Bernoulli distribution: $m \sim \text{Bernoulli}(\Phi(x))$. The input $\Phi(x)$ of the distribution is given by the cumulative distribution function of the standard normal distribution, namely $\Phi(x) = P(X \leq x), X \sim \mathcal{N}(0,1)$. The standard normal distribution is chosen because neuron input values generally follow a normal distribution. Finally, as the multiplication of x by m is stochastic and the output should be deterministic, the expected value is calculated in order to find the final formula of GELU:

$$\text{GELU}(x) = \mathbb{E}[mx] = x\mathbb{E}[m] = x\Phi(x) = x\frac{1}{2}[1 + \text{erf}(x/\sqrt{2})]. \tag{4.3}$$

Figure 4.1 shows the behavior of the GELU function against the behavior of the common activation functions ReLU and ELU.

The effectiveness of GELU against other famous activation functions has been proven by [41] for the MNIST and TIMIT datasets. In this section, the effectiveness of the GELU function is tested against the effectiveness of the ReLU and ELU activation functions. Three different experiments have been conducted under the same set of hyperparameters, but three different activation functions have been employed for the GEM. Figure 4.2, shows the test set approximation ratio, whose precise meaning is elucidated in Chapter 7, for MIS as the model is being trained. The figure shows the ability of the model to learn quicker if a GELU function is employed against a ReLU and an ELU.

4.3 $M_{\boldsymbol{\theta}}$ model: the final overview

As mentioned before, the second module of the model architecture is composed of a standard fully connected neural network [43], as depicted by the right part of Figure 4.3. The input to the FCNN is the graph embedding $\boldsymbol{\mu}_G$ obtained by the GEM. Once the input is propagated to the network, it reaches the final layer that

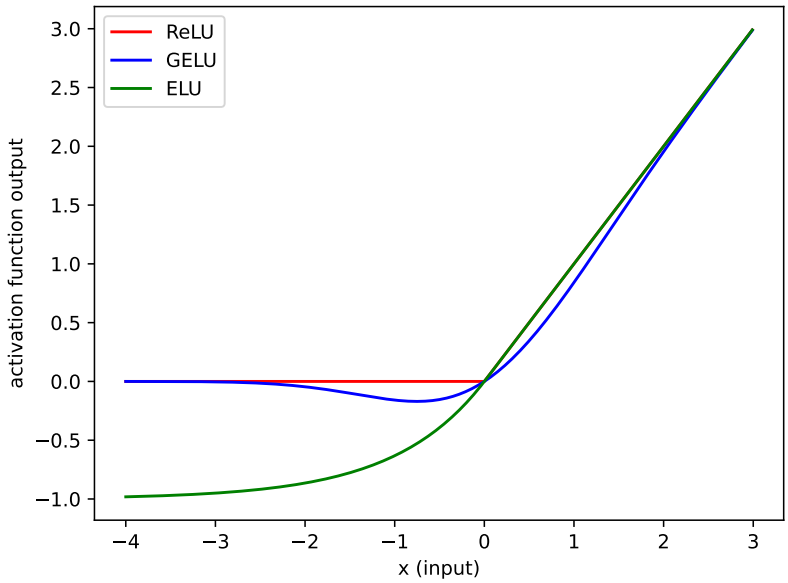

Figure 4.1: The plot shows the behavior of three curves produced by three distinct activation functions.

outputs the graph logit value l_G, associated with the graph G given as input to $M_{\boldsymbol{\theta}}$.

The final structure of $M_{\boldsymbol{\theta}}$ is highlighted in Figure 4.3. Initially, an input graph G is passed to the model with zeros as node embeddings, which are displayed as white in the figure. After K iterations of the GEM module, the final node embeddings are obtained. These are then averaged to obtain a graph embedding $\boldsymbol{\mu}_G$. Finally, the graph embedding is put through multiple fully connected layers to obtain a final logit value for the input graph.

During the training process, the comparator function $\text{CMP}(G_0, G_1)$ uses the result of $M_{\boldsymbol{\theta}}(G_0)$ and $M_{\boldsymbol{\theta}}(G_1)$ for training the model itself, as described later by Chapter 5. Unfortunately, the logit value outputted by $M_{\boldsymbol{\theta}}(G)$ is not a number in (0,1) and, thus, it does not represent a probability. For this reason, a softmax function is employed. This function is a differentiable loss function that takes the logit values $M_{\boldsymbol{\theta}}(G_0), M_{\boldsymbol{\theta}}(G_1)$ and outputs the following probability binary vector:

$$\mathbf{p} = \text{softmax}([M_{\boldsymbol{\theta}}(G_0) \| M_{\boldsymbol{\theta}}(G_1)]). \tag{4.4}$$

The probability binary vector calculated as 4.4 is employed for classification. If

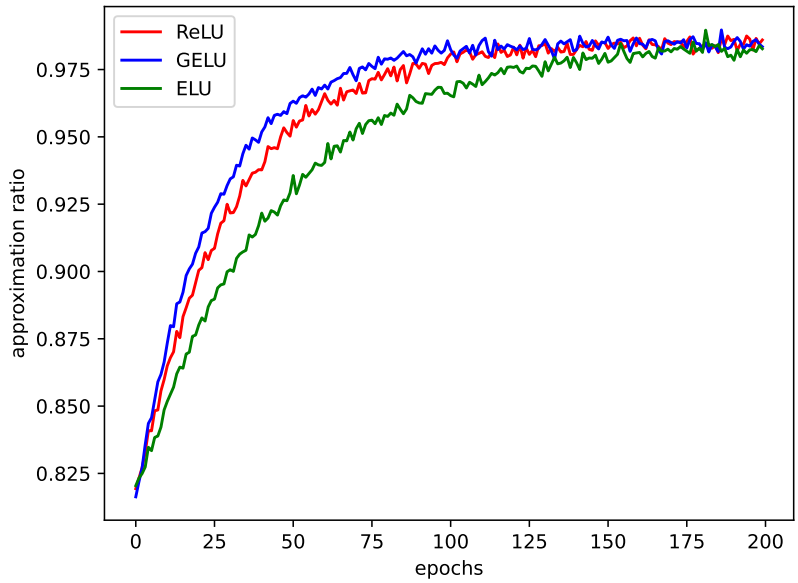

Figure 4.2: The figure shows the test set approximation ratio (the higher the better) for MIS over the COLLAB dataset. Each curve has been obtained with the same hyperparameters but different activation functions.

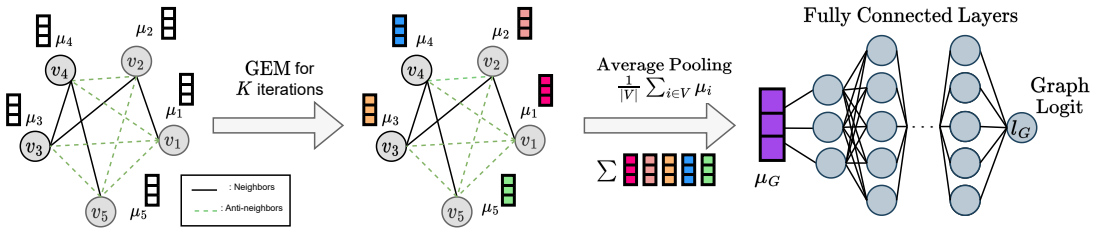

Figure 4.3: Architecture of model $M_{\boldsymbol{\theta}}(G)$, including the GEM module and the fully connected layers. The striped green edges of the GEM module connect the anti-neighbors, while the black edges connect the neighbors. The pile above each node v_i represents the node embedding $\boldsymbol{\mu}_i$. Every node embedding is then averaged ($\sum$ symbol in the figure) to obtain the graph embedding $\boldsymbol{\mu}_G$, represented by the purple pile.

$(\mathbf{p})_0 > (\mathbf{p})_1$ then the comparator chooses G_0, otherwise it branches over G_1.

Chapter 5

Training the model $M_{\boldsymbol{\theta}}$

The ML model $M_{\boldsymbol{\theta}}$, whose architecture is described in Section 4, is employed by the proposed algorithms in order to find a solution to the MIS and MVC problems. Unfortunately, as later described by Section 5.1, training the ML model with a classic supervised approach is not efficient at all. Thus, this work utilizes a self-supervised approach based on the theoretical property denoted as *consistency property*. Thanks to this property and thanks to a smart data generation process, described in Section 5.2, it is possible to efficiently train the model $M_{\boldsymbol{\theta}}$.

5.1 Model training: why not supervised?

The most straightforward approach for training the model $M_{\boldsymbol{\theta}}$ would be a supervised approach. A classic supervised training would select the parameters $\boldsymbol{\theta} \in \Theta$ such that $\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}(G_0, G_1) \simeq \mathbb{I}\left[|\text{MIS}(G_0)| < |\text{MIS}(G_1)|\right]$ for the MIS problem or, in case of the MVC problem, $\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}(G_0, G_1) \simeq \mathbb{I}\left[|\text{MVC}(G_0)| > |\text{MVC}(G_1)|\right]$.
Unfortunately, a supervised approach would require a huge amount of annotated data of the form $\{((G_0, G_1), \mathbb{I}\left[|\text{MIS}(G_0)| < |\text{MIS}(G_1)|\right])\}$ for MIS and, considering the MVC problem, of the form $\{((G_0, G_1), \mathbb{I}\left[|\text{MVC}(G_0)| > |\text{MVC}(G_1)|\right])\}$. As mentioned in Section 2.1.4, the MIS and the MVC problems are NP-Hard, thus annotating such data comes with an insurmountable computational burden.

The key idea to overcome the latter limitation is to annotate the data of the form $\{(G_0, G_1)\}$ by using the algorithm $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}$ for MIS and the algorithm $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}$ for MVC. The biggest improvement introduced by the usage of the algorithms with respect to the classic annotated data is in the computational complexity. Indeed,

the proposed algorithms of this work run in polynomial time with respect to the size of the graph. Intuitively, the proposed framework entails the optimization of the parameterized comparator functions $\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}$ and $\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}$ on data generated using algorithms $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}$ and $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}$ respectively for the MIS and MVC problems. A better comparator function leads to a better algorithm, which leads to better data, and vice versa. This mutually reinforcing relationship, which is at the basis of the self-supervised training, between the two components of the framework is theoretically indicated by Theorems 3,4 that are presented in Sections 5.1.1,5.1.2.

5.1.1 Consistency comparator for MIS

At the heart of the self-supervised learning approach for MIS lies the knowledge of *consistent comparator for MIS*. In particular, a comparator function for MIS $\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}} : \mathcal{G} \times \mathcal{G} \mapsto \{0,1\}$ is called consistent if and only if for any pair of graphs $G_0, G_1 \in \mathcal{G}$ the following holds:

$$\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}(G_0, G_1) = 0 \text{ if and only if } \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_0)\right|\right] \geq \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_1)\right|\right]. \tag{5.1}$$

Remark 7. *In Equation 5.1 the expressions* $\mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_0)\right|\right], \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_1)\right|\right]$ *are used since, as already discussed, a comparator* $\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}$ *for MIS induces a* ***randomized*** *algorithm* $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}$ *for MIS.*

Equation 5.1 highlights that a consistent comparator for MIS induces an algorithm that produces, under expected value, solutions that are coherent with respect to the choice made by the comparator. Thus, if a consistent comparator chooses, as an example, a graph G_0 among two graphs G_0 and G_1, then the induced algorithm coherently states that the MIS of G_0 is greater than the MIS of G_1, thus confirming the choice of the consistent comparator. Moreover, the opposite implications also apply. Indeed, given, for example, an algorithm induced by a consistent comparator stating that a graph G_0 has a higher MIS than a graph G_1, then the algorithm leverages a comparator that chooses G_0 among the two graphs G_0 and G_1.

At this point, one good question that arises is the reason why a consistent comparator would be important. The answer to this question is given by the following theorem:

Theorem 3. *Given a consistent comparator for MIS* $\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}} : \mathcal{G} \times \mathcal{G} \mapsto \{0,1\}$*, the induced algorithm* $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}$ *always computes a Maximum Independent Set,* $\mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G)\right|\right] = |\text{MIS}(G)|$ *for all* $G \in \mathcal{G}$*.*

Proof. Consider a consistent comparator for MIS $\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}$. Theorem 3 will be proven with an induction on the number of edges i.

- *Induction Basis ($i = 0$):* Let $G(V, E) \in \mathcal{G}[0]$ then $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G) = V = \text{MIS}(G)$.
- *Induction Hypothesis*: $\mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G)\right|\right] = |\text{MIS}(G)|$ for all $G \in \mathcal{G}[j]$ with $j \leq i$.
- *Induction Step*: Let $G \in \mathcal{G}[i+1]$ and consider a node v with degree $d(v) \geq 1$. Consider also the graphs $G_0 := G/\{v\}$ and $G_1 := G/\{\mathcal{N}(v)\}$. Both G_0 and G_1 admit less than i edges and thus by the inductive hypothesis, $\mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_0)\right|\right] = |\text{MIS}(G_0)|$ and $\mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_1)\right|\right] = |\text{MIS}(G_1)|$. Hence $\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}(G_0, G_1) = 0$ if and only if $|\text{MIS}(G_0)| \geq |\text{MIS}(G_1)|$. As a result, $\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G)\right| = \max\left(|\text{MIS}(G_0)|, |\text{MIS}(G_1)|\right)$ and Theorem 1 implies that $\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G)\right| = |\text{MIS}(G)|$.

□

Theorem 3 is very important. It establishes that a consistent comparator for MIS induces an optimal algorithm. So, at this point, the goal of training for MIS is to find the set of parameters $\boldsymbol{\theta}^\star \in \Theta$ such that the resulting comparator $\text{CMP}_{\boldsymbol{\theta}^\star}^{\text{MIS}}$ is consistent:

$$\text{CMP}_{\boldsymbol{\theta}^\star}^{\text{MIS}} = 0 \;\; \textit{if and only if} \;\; \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}^\star}^{\text{MIS}}}(G_0)\right|\right] \geq \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}^\star}^{\text{MIS}}}(G_1)\right|\right] \; . \tag{5.2}$$

By definition of a consistent comparator for MIS, in order to train a comparator effectively what is needed is the output of the induced algorithm (see Equation 5.1). This permits us to overcome the NP-hardness of the MIS problem, as the computational complexity of the proposed algorithm of this work is polynomial.

5.1.2 Consistency comparator for MVC

Similarly to what was discussed in Section 5.1.1, the idea of *consistent comparator for MVC* is crucial in the self-supervised learning approach of the comparator for MVC. Given a comparator function for MVC $\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}} : \mathcal{G} \times \mathcal{G} \mapsto \{0{,}1\}$, it is called consistent if and only if for any pair of graphs $G_0, G_1 \in \mathcal{G}$ the following holds:

$$\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}(G_0, G_1) = 0 \textit{ if and only if } \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_0)\right|\right] \leq \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_1)\right|\right] \; . \tag{5.3}$$

Remark 8. *In Equation 5.3 the expressions* $\mathbb{E}\left[\left|\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}}(G_0)\right|\right], \mathbb{E}\left[\left|\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}}(G_1)\right|\right]$ *are used since, as already discussed, a comparator* $\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}$ *for MVC induces a* ***randomized*** *algorithm* $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}}$ *for MVC.*

The same reasoning made for the MIS problem in Section 5.1.1 can be made for the MVC problem. Indeed, Equation 5.3 underscores the idea that when a comparator for MVC is consistent, the algorithm it induces will, on average, yield solutions that align with the comparator's choices. For instance, if a consistent comparator for MVC opts for one graph, say G_0, over two options G_0 and G_1, the resulting induced algorithm will assert that the MVC of G_0 is smaller than that of G_1, thus affirming the comparator's choice. Conversely, the converse holds true as well. In other words, if an algorithm, arising from a consistent comparator for MVC, indicates that the MVC of a graph G_0 is smaller than that of G_1, it implies that the comparator favored G_0 when choosing between the two graphs G_0 and G_1.

Equivalently to the MIS case, a consistent comparator for MVC induces an algorithm for MVC that is optimal:

Theorem 4. *Let a consistent comparator for MVC* $\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}} : \mathcal{G} \times \mathcal{G} \mapsto \{0,1\}$. *Then the induced algorithm* $\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}}$ *always computes a Minimum Vertex Cover,* $\mathbb{E}\left[\left|\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}}(G)\right|\right] = |\mathrm{MVC}(G)|$ *for all* $G \in \mathcal{G}$.

The proof of Theorem 4 is omitted as it is almost identical to the proof of Theorem 3.

Thanks to Theorem 4, the goal is to train a comparator for MVC in order to make it as consistent as possible. According to Equation 5.3, this can be done by leveraging the results of the induced algorithm, that runs polynomially. Then, as well as the MIS problem, the goal of training for the MVC comparator $\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}}$ is to find the set of parameters $\boldsymbol{\theta}^\star \in \Theta$ such that the comparator is consistent:

$$\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}^\star} = 0 \quad \textit{if and only if} \quad \mathbb{E}\left[\left|\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}^\star}}(G_0)\right|\right] \leq \mathbb{E}\left[\left|\mathcal{A}^{\mathrm{CMP}^{\mathrm{MVC}}_{\boldsymbol{\theta}^\star}}(G_1)\right|\right] . \quad (5.4)$$

5.2 Training process

The cornerstone idea of the self-supervised learning approach is to make the comparator more and more consistent over time. Namely, the idea is to update the parameters, respectively for the MIS and MVC comparators, as follows:

$$\boldsymbol{\theta}_{t+1} := \mathrm{argmin}_{\boldsymbol{\theta}\in\Theta}\mathbb{E}_{G_0,G_1}\left[\ell\left(\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}}(G_0,G_1), \mathbb{I}\left[\mathbb{E}\left[\left|\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}_t}}(G_0)\right|\right] < \mathbb{E}\left[\left|\mathcal{A}^{\mathrm{CMP}^{\mathrm{MIS}}_{\boldsymbol{\theta}_t}}(G_1)\right|\right]\right]\right)\right]. \quad (5.5)$$

Algorithm 3 Basic Pipeline of the Self-Supervised Training Approach

1: **Input:** A distribution $\mathcal{D}_{\text{train}}$ over graphs.
2: Initialize parameters $\boldsymbol{\theta}_0 \in \Theta$.
3: Initialize $\mathcal{S}_{\text{train}} = \mathcal{D}_{\text{train}}$.
4: **for** each super-epoch $t^{\text{super}} = 0, \dots, T^{\text{super}} - 1$ **do**
5: Pick randomly R_{tot} graphs $[G_{\text{init}}^{(1)}, G_{\text{init}}^{(2)}, \dots, G_{\text{init}}^{(i)}, \dots, G_{\text{init}}^{(R_{\text{tot}})}] \sim \mathcal{S}_{\text{train}}$.
6: Update $\mathcal{S}_{\text{train}}$ as $\mathcal{S}_{\text{train}} = \mathcal{S}_{\text{train}} - [G_{\text{init}}^{(1)}, G_{\text{init}}^{(2)}, \dots, G_{\text{init}}^{(i)}, \dots, G_{\text{init}}^{(R_{\text{tot}})}]$
7: **for** each graph $G_{\text{init}}^{(i)}$ **do**
8: **Recursive tree**: Run $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}_t}}(G_{\text{init}}^{(i)})$ and gather the leaf graphs
9: **Roll-outs**: Estimate the MIS/MVC of each leaf graph
10: **Pairwise samples**: Construct datasamples by coupling each leaf node
11: **end for**
12: **for** each sub-epoch $t = 0, \dots, T^{\text{sub}} - 1$ **do**
13: Update the parameters $\boldsymbol{\theta}_{t+1} \in \Theta$ as in Equation 5.5 or 5.6
14: **end for**
15: **end for**

$$\boldsymbol{\theta}_{t+1} := \text{argmin}_{\boldsymbol{\theta}\in\Theta} \mathbb{E}_{G_0,G_1}\left[\ell\left(\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}(G_0, G_1), \mathbb{I}\left[\mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}_t}^{\text{MVC}}}(G_0)\right|\right] > \mathbb{E}\left[\left|\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}_t}^{\text{MVC}}}(G_1)\right|\right]\right]\right)\right]. \tag{5.6}$$

where $\ell(\cdot, \cdot)$ is a binary classification loss.

Algorithm 3 summarizes the training pipeline. For each super-epoch, R_{tot} graphs are picked from the set of graphs $\mathcal{S}_{\text{train}}$. The $\mathcal{S}_{\text{train}}$ is updated in Line 6 in order to avoid picking the same graphs in the next super-epoch. Then, for each graph, three important steps are executed. The *recursive tree* step, described in Section 5.2.1, computes the leaf graphs by running the induced algorithm. Then, in the *roll-out* step elucidated in Section 5.2.2, an estimate of either the MIS or MVC is evaluated for each leaf graph. The last step, denoted as *pairwise samples* (Section 5.2.3), consists of coupling all leaf graphs in order to produce datasamples. Line 13 of the algorithm shows the update of the model parameters, performed thanks to the datasamples generated in the previous for-loop.
It is important to point out the different uses of the super-epochs and the sub-epochs. During each super-epoch, a set of datasamples is generated. Then, for every sub-epoch the same set of datasamples is employed for updating the model parameters.

Algorithm 3 is the training pipeline for both the MIS and MVC comparators. For this reason, the comparator symbol in line 6 of Algorithm 3 does not have either the superscript MIS or the superscript MVC.

5.2.1 Recursive tree

As described in Chapter 3, the proposed algorithms are recursive, and two graphs are produced for each iteration of the algorithms. Thus, all the graphs generated at each iteration can be represented by a tree, denoted as "recursive tree", where each level i of the tree corresponds to iteration i of the algorithm and contains two nodes, which are the graphs generated in iteration i. Figure 5.1 shows a specific example of the recursive tree step starting from a graph G_{init}. In the figure, every leaf of the tree is a graph called "leaf graph". The red leaf graphs of the figure, namely G_2, G_3, G_6, are the graphs chosen by the comparator, the ones that are not chosen, namely G_1, G_4, G_5, are in gray. Moreover, in the example displayed by Figure 5.1, the leaf graph G_6 is drawn with a red circle, as the algorithm underwent the ending step.
At the end, all leaf graphs are gathered and employed in the subsequent step (Section 5.2.2). They are the starting point for the generation of the datasamples.

5.2.2 Roll-out

The recursive tree step considers a graph as input and outputs several leaf graphs. Then, each leaf graph undergoes the roll-out step. The idea of this procedure is taken from the "simulation" step of the Monte Carlo Tree Search of Section 2.2.2. Indeed, similar to the MCTS, the roll-out step aims at estimating the value of each node of the recursive tree, performing an estimate of the MIS or the MVC (depending on the analyzed problem) of each leaf graph. The difference between the simulation of the MCTS and the roll-out step of the proposed algorithms lies in the way the estimation is performed. In the case of the MCTS, the estimate is calculated by making random moves until a terminal state is reached. In the roll-out step of the proposed algorithms, the estimate is not randomly generated but calculated by the algorithm induced by the comparator that is being trained.

In the case of the algorithm for MIS, the estimate of the roll-out step is made by running the algorithm induced by the comparator M_{tot} times for every leaf graph. Then, the estimated MIS value $EV_{G_i}^{\text{MIS}}$ corresponding to leaf graph G_i is

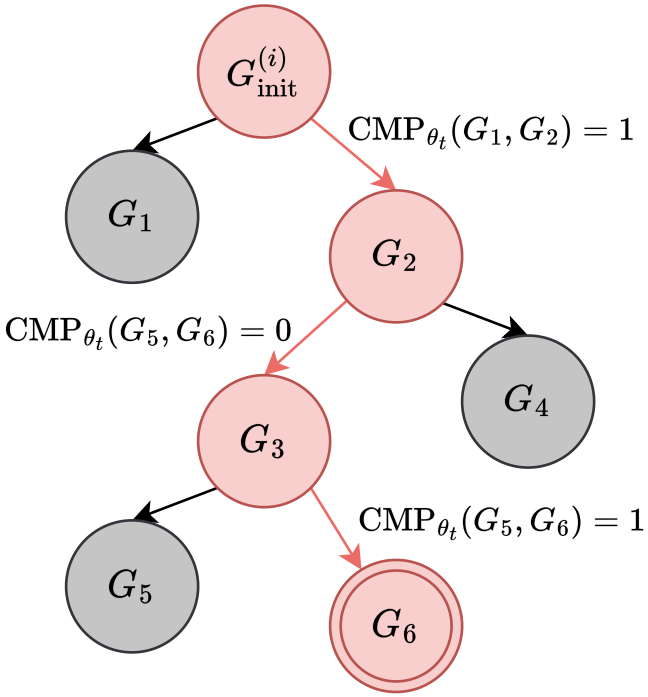

Figure 5.1: Recursive tree step: Starting from a graph $G^{(i)}_{\text{init}}$, the induced algorithm generates the leaf graphs $G_1, G_2, G_3, G_4, G_5, G_6$. The red nodes of the tree represent the leaf graphs chosen by the comparator, while the gray nodes are the remaining leaf graphs. The leaf graph G_6 is represented by a node that has a thin red circle, as the algorithm underwent the ending step. Moreover, it is worth noticing that the comparator symbols $\text{CMP}_{\boldsymbol{\theta}_t}$ do not have the superscript MIS or MVC, since the image does not specifically refer to either the comparator for MIS or for MVC.

the maximum between the M_{tot} runs of the algorithm:

$$\begin{aligned}
EV^{\text{MIS}}_{G_1} &= \max\left(|(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_1))^{(1)}|, |(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_1))^{(2)}|, \ldots, |(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_1))^{(M_{\text{tot}})}|\right),\\
EV^{\text{MIS}}_{G_2} &= \max\left(|(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_2))^{(1)}|, |(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_2))^{(2)}|, \ldots, |(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_2))^{(M_{\text{tot}})}|\right),\\
&\vdots\\
EV^{\text{MIS}}_{G_i} &= \max\left(|(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_i))^{(1)}|, |(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_i))^{(2)}|, \ldots, |(\mathcal{A}^{\text{CMP}^{\text{MIS}}_{\boldsymbol{\theta}}}(G_i))^{(M_{\text{tot}})}|\right),\\
&\vdots
\end{aligned} \tag{5.7}$$

The roll-out step for the algorithm for MVC is similar to the one for MIS. The only difference is that the minimum, not maximum, among the M_{tot} runs is the

estimated value:

$$
\begin{aligned}
EV_{G_1}^{\text{MVC}} &= \min\left(|(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_1))^{(1)}|, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_1))^{(2)}|, \ldots, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_1))^{(M_{\text{tot}})}|\right),\\
EV_{G_2}^{\text{MVC}} &= \min\left(|(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_2))^{(1)}|, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_2))^{(2)}|, \ldots, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_2))^{(M_{\text{tot}})}|\right),\\
&\vdots\\
EV_{G_i}^{\text{MVC}} &= \min\left(|(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_i))^{(1)}|, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_i))^{(2)}|, \ldots, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_i))^{(M_{\text{tot}})}|\right),\\
&\vdots
\end{aligned}
\tag{5.8}
$$

As shown by Algorithm 3, the roll-out step is executed several times for each super-epoch. Naturally, the estimates calculated during the first super-epochs are not good, since the comparator has not been trained enough yet. Conversely, as soon as the comparator is being trained super-epoch after super-epoch, the estimates get better and better.

5.2.3 Pairwise samples

The pairwise samples step receives several leaf graphs, and each leaf graph is associated with an estimate. The last step consists of creating datasamples from all possible couples of leaf graphs. As an example, if the total number of leaf graphs is 100, then the total number of couples, and thus datasamples, is $\binom{100}{2}$.

A datasample is a couple of graphs G_i, G_j associated with a number that can be either 0 or 1. In the case of the comparator for MIS, the number is 0 if the estimate for G_i is greater than the estimate for G_j, namely $EV_{G_i}^{MIS} > EV_{G_j}^{MIS}$, otherwise is 1. Instead, if the comparator for MVC is being trained, the number is 0 is $EV_{G_i}^{MVC} < EV_{G_j}^{MVC}$ and 1 otherwise. The above description is summarized by the following equation that shows the structure of a datasample for the MIS and MVC comparators respectively:

$$\langle (G_i, G_j), \mathbb{I}[EV_{G_i}^{MIS} < EV_{G_j}^{MIS}] \rangle. \tag{5.9}$$

$$\langle (G_i, G_j), \mathbb{I}[EV_{G_i}^{MVC} > EV_{G_j}^{MVC}] \rangle. \tag{5.10}$$

Figure 5.2 shows a specific example of the three steps of Sections 5.2.1,5.2.2,5.2.3. Starting from a graph $G_{\text{init}}^{(i)}$, 6 leaf graphs are generated by the proposed algorithm thanks to the comparator's choices. Then, the 6 leaf graphs undergo the roll-out step, and each graph is associated with an estimate. Finally, in the pairwise-samples step, $\binom{6}{2} = 15$ pairs are formed by coupling the 6 leaf graphs. Every couple is then associated with a number ($\mathbb{I}[EV_{G_i} \lesseqgtr EV_{G_j}]$ in Figure 5.2) to form a datasample.

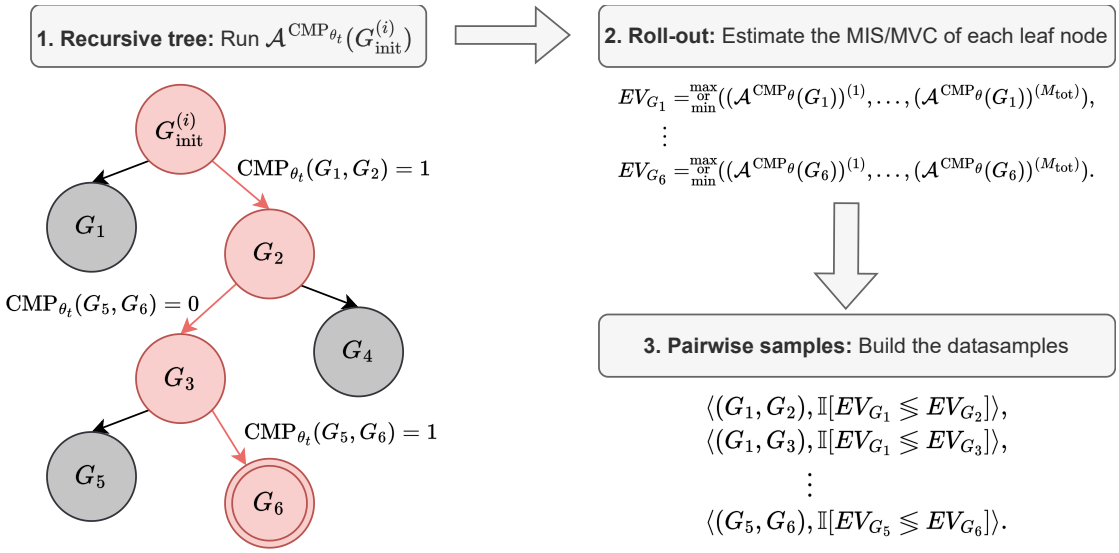

Figure 5.2: The figure illustrates the three steps outlined in Sections 5.2.1,5.2.2,5.2.3 through which a graph denoted as $G^{(i)}_{\mathrm{init}}$ undergoes. Notably, the symbols for the comparator and the estimated value lack the superscripts MIS or MVC, and the symbol $\lesseqgtr$ replaces the symbols $<$ or $>$. Additionally, the notation $\overset{\max}{\underset{\min}{\text{ or }}}$ is employed in step 2. These choices are made because the training algorithm depicted in the figure does not explicitly refer to either the MIS or MVC problem, but it is a general framework valid for both problems.

Chapter 6

Model and algorithmic enhancements

The basic operating principles of the proposed algorithms, the model architecture, and the training process are described in the previous chapters. This chapter focuses on the modifications that have been proposed in order to improve the final results. Particularly, Sections 6.2,6.4 describe changes to the model architecture, Section 6.1 shows a modification to the training process, Section 6.5 elucidates a modification in the comparator usage, and Section 6.3 describes how the dataset augmentation operation is performed in this work.

6.1 Mixed Roll-Outs

Among the modifications described in this work, the *mixed roll-outs* one is probably the most effective. The idea is to improve the roll-out step of Section 5.2.2 by incorporating the result of a greedy algorithm. The estimates of Section 5.2.2 are evaluated as the maximum or minimum (depending on MIS or MVC) among M_{tot} runs. This means that the estimate is the "best" result among the M_{tot} runs. By incorporating the result of a greedy algorithm that is totally external with respect to the proposed algorithms, the estimate EV_{G_i} can improve. The greedy algorithm, later described in Chapter 7, has the advantage of producing solutions that are often near-optimal, and such solutions are found incredibly fast. Naturally, two distinct greedy algorithms are employed for the MIS and MVC problems, but the operating principle is the same for both of them.

The addition of the greedy result, brought by the mixed roll-outs modification, changes the Equations 5.7,5.8 into, respectively, the following equations:

$$EV_{G_i}^{\text{MIS}} = \max\left(|\text{Greedy}^{\text{MIS}}(G_i)|, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_i))^{(1)}|, \ldots, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MIS}}}(G_i))^{(M_{\text{tot}})}|\right). \tag{6.1}$$

$$EV_{G_i}^{\text{MVC}} = \min\left(|\text{Greedy}^{\text{MVC}}(G_i)|, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_i))^{(1)}|, \ldots, |(\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}^{\text{MVC}}}(G_i))^{(M_{\text{tot}})}|\right). \tag{6.2}$$

6.2 Concatenation Model

The model $M_{\boldsymbol{\theta}}$ described in Chapter 4, used by the comparator for deciding between two graphs G_0, G_1 (see Equations 3.2,3.3), takes only one graph as input. This implies that the graphs G_0 and G_1 are treated independently from one another. The result of $M_{\boldsymbol{\theta}}(G_0)$ is not influenced at all by the result of $M_{\boldsymbol{\theta}}(G_1)$. The *concatenation model* modification aims at introducing interdependence among the two graphs G_0, G_1. This is done by modifying the model of Chapter 4 by implementing a new architecture $M'_{\boldsymbol{\theta}}$. Differently from $M_{\boldsymbol{\theta}}$, the ML model $M'_{\boldsymbol{\theta}}$ is defined as $M'_{\boldsymbol{\theta}}(G_0, G_1) : \mathcal{G} \times \mathcal{G} \mapsto \mathbb{R}^2$ and, consequently, takes two graphs as input and outputs a bi-dimensional vector. The vector is a probability binary vector that indicates the probability of each graph. Then, the comparator $\text{CMP}_{\boldsymbol{\theta}}(G_0, G_1)$ chooses the graph associated with the highest probability:

$$\begin{aligned} \mathbf{p} &= M'_{\boldsymbol{\theta}}(G_0, G_1), \\ \text{CMP}_{\boldsymbol{\theta}}(G_0, G_1) &= \text{argmax}((\mathbf{p})_0, (\mathbf{p})_1). \end{aligned} \tag{6.3}$$

Figure 6.1 shows the architecture of the concatenated model. Two graphs G_0, G_1 are given to two distinct GEMs (see Section 4.1 for the GEM description). The resulting graph embeddings are concatenated and given as input to a fully connected neural network that outputs a probability binary vector. Since the embeddings are concatenated, their mutual interdependence is leveraged by the FCNN.

6.3 Dataset Augmentation

Normally, the training graphs employed during the training process are randomly selected from the distribution $\mathcal{D}_{\text{train}}$ while the model is being trained. Thus, the R_{tot} graphs of Line 4 of Algorithm 3 have random sizes. What if the training

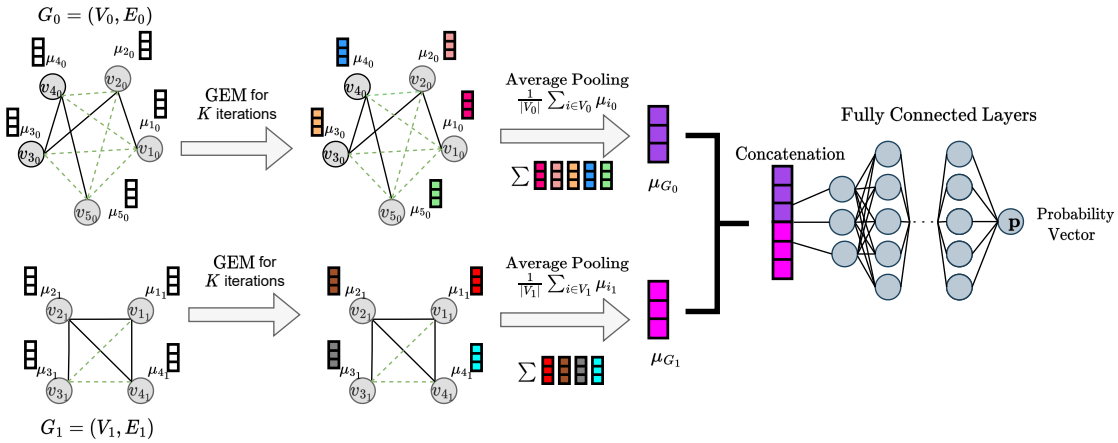

Figure 6.1: The concatenation model displayed in the figure takes two graphs G_0, G_1 as input. The graphs are passed to two distinct GEM modules and the resulting embeddings ($\boldsymbol{\mu}_{G_0}$ and $\boldsymbol{\mu}_{G_1}$ in the figure) are concatenated and passed to an FCNN. The output of the model is a binary probability vector.

graphs from $\mathcal{D}_{\text{train}}$ are sorted in terms of the total number of nodes? The model would be first trained over small graphs and, as soon as the training process goes on, graphs of larger and larger size would be used for training the comparator. Thanks to this sorting approach, when the model is being trained over large graphs, it is already "initialized" over the small graphs and can, in principle, learn much better from the big training graphs.

Unfortunately, the sorting approach on its own does not work. Indeed, once the comparator is being trained over the big graphs it seems to forget what it learned in the past, and thus the induced algorithm does not provide any more good solutions for the test graphs of small size, resulting in a worse test set approximation ratio[1]. What is mentioned above can be noted in Table 6.1. The table shows the test set approximation ratio for the MIS problem and the COLLAB dataset as soon as more and more training graphs were used for training. The row *Base* refers to the model being normally trained (no sorting), while row *Pure Sorting* shows the results for the purely sorted dataset. It can be noted that the test set approximation ratio for MIS of the row "pure sorting" drops as soon as the 80% of the graphs are trained. This is a direct effect of what was mentioned above: when the model is being trained over larger graphs, the results get worse.

To avoid a drop in the performances in the case of the purely sorted dataset,

[1]See Chapter 7 for a precise explanation of what the test set approximation ratio is.

Table 6.1: Test set approximation ratio for MIS (the higher the better) during the training process for the COLLAB dataset using different methods. The results of every column are obtained after the model is trained over a specific number of graphs (in percentage). Every row refers to a specific method that is used.

Method (↓) Training graphs % (→)	20%	40%	60%	80%	100%
Base (no sorting)	0.912	0.953	0.979	0.986	0.990
Pure Sorting	0.961	0.976	0.982	0.953	0.951
Dataset Augmentation	0.959	0.977	0.988	0.990	0.996

a smart dataset augmentation is performed. In particular, the purely sorted dataset is augmented by adding smaller graphs between the larger ones. Once the model is trained over the larger graphs, it is also re-trained over some of the smaller ones and it does not forget what it learned in the past. The results of the dataset augmentation modification are shown in row *Dataset Augmentation* of Table 6.1. Surprisingly, the model performances do not drop, and the final test set approximation ratio for the dataset augmentation, after the 100% of the train graphs were used, is even better than the one for the normally trained model.

6.4 Layer Augmentation

As mentioned in Section 6.3, training the model with a pure sorted dataset is in principle a good idea: while the model is being trained over the large graphs, it has already been "initialized" as it already learned from small graphs. Unfortunately, Section 6.3 shows in Table 6.1 that this approach does not work on its own, and a countermeasure has to be taken. The modification *Layer Augmentation* proposed in this section is a specific countermeasure for the model trained with the purely sorted dataset. The idea is to enhance the model's capabilities as graphs of bigger and bigger sizes are employed in the training process. This is performed by adding one macro-layer MacL_{K+1} at the end of the GEM module as soon as the training process goes on. If, before the augmentation, the model had K macro-layers for the GEM module, the augmented model will have $K+1$ macro-layers

In Figure 6.2, the GEM module of a model $M_{\boldsymbol{\theta}}$ goes from $K = 3$ macro-layers to $K = 4$ macro-layers. The parameters of the macro-layers $MacL_0^{\text{new}}, MacL_1^{\text{new}}, MacL_2^{\text{new}}$ of the augmented GEM module are the same as the old GEM module.

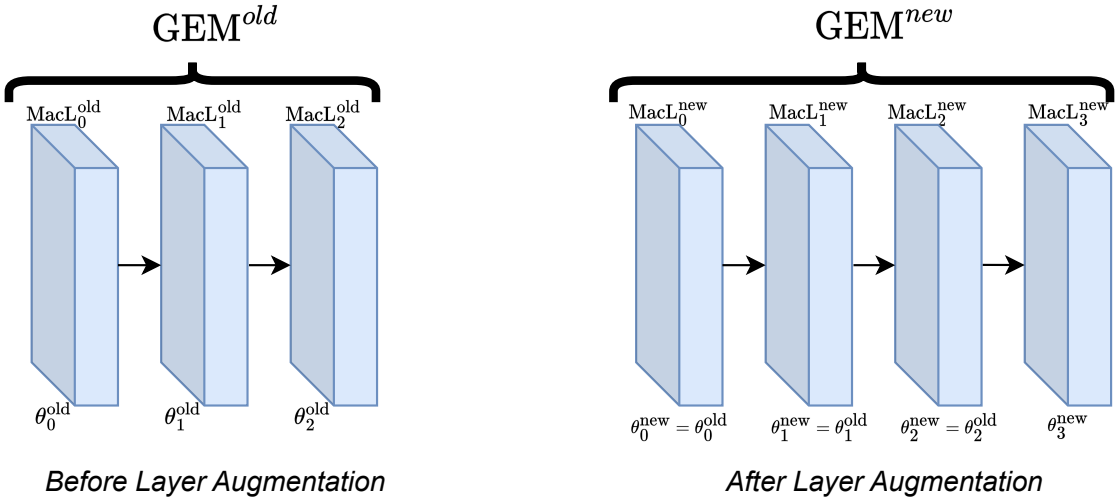

Figure 6.2: The figure shows the GEM module of a model $M_{\boldsymbol{\theta}}$ before and after the augmentation by one macro-layer. The model parameters of the macro-layers $MacL_0^{\text{new}}, MacL_1^{\text{new}}, MacL_2^{\text{new}}$ of the new GEM module are the same as the old GEM module, except for, obviously, the model parameters of the last macro-layer $MacL_3^{\text{new}}$. The figure does not display the activation functions and the normalization layers.

6.5 Ensemble Learning

Before delving into the description of the modification, it has to be remarked that the comparator symbols $\text{CMP}_{\boldsymbol{\theta}}$ displayed in this section do not contain the superscripts MIS and MVC, as what is written in this section refers to both the algorithms for MIS and MVC.

One of the most important states of this work is that a comparator $\text{CMP}_{\boldsymbol{\theta}}$ induces an algorithm $\mathcal{A}^{\text{CMP}_{\boldsymbol{\theta}}}$. Then, the comparator that is designed by an ML model can be properly trained such that the induced algorithm finds solutions that are near-optimal. The *Ensemble Learning* modification does not use a single comparator $\text{CMP}_{\boldsymbol{\theta}}$ but, instead, multiple comparators $[\text{CMP}_{\boldsymbol{\theta}^{(1)}}, \text{CMP}_{\boldsymbol{\theta}^{(2)}}, \ldots, \text{CMP}_{\boldsymbol{\theta}^{(L)}}]$ induce one single algorithm, denoted as *Ensemble Algorithm*. The idea of this modification is that a comparator does not deal with all possible kinds of graphs, but each comparator $\text{CMP}_{\boldsymbol{\theta}^{(i)}}$ handles graphs having a specific number of nodes. To this

end, each $\text{CMP}_{\boldsymbol{\theta}^{(i)}}$ is assigned an interval of nodes:

$$\begin{aligned}
\text{CMP}_{\boldsymbol{\theta}^{(1)}} &\longrightarrow [1, N_{\text{max}}^{(1)}), \\
\text{CMP}_{\boldsymbol{\theta}^{(2)}} &\longrightarrow [N_{\text{min}}^{(2)}, N_{\text{max}}^{(2)}), \\
&\vdots \\
\text{CMP}_{\boldsymbol{\theta}^{(i)}} &\longrightarrow [N_{\text{min}}^{(i)}, N_{\text{max}}^{(i)}), \\
&\vdots \\
\text{CMP}_{\boldsymbol{\theta}^{(L)}} &\longrightarrow [N_{\text{min}}^{(L)}, \infty).
\end{aligned} \tag{6.4}$$

It is important to remark that the maximum interval boundary of a range coincides with the minimum interval boundary of the subsequent range, namely: $N_{\text{max}}^{(i)} = N_{\text{min}}^{(i+1)}$.

Once the range-of-nodes assignment is performed, for each iteration, the ensemble algorithm chooses one comparator among the list $[\text{CMP}_{\boldsymbol{\theta}^{(1)}}, \ldots, \text{CMP}_{\boldsymbol{\theta}^{(L)}}]$ in order to decide between two graphs $G_0 = (V_0, E_0)$ and $G_1 = (V_1, E_1)$. The choice of the comparator is made according to the number of nodes $|V_0|, |V_1|$ of the two graphs. Specifically, the average number of nodes $N_{\text{avg}} = (|V_0| + |V_1|)/2$ value is computed. Then, the ensemble algorithm checks on which interval $[N_{\text{min}}^{(i)}, N_{\text{max}}^{(i)})$ of Equation 6.4 does the value N_{avg} fall. The comparator $\text{CMP}_{\boldsymbol{\theta}^{(i)}}$ corresponding to $[N_{\text{min}}^{(i)}, N_{\text{max}}^{(i)})$ is the one used by the ensemble algorithm.

The training process for the ensemble algorithm, as depicted in Algorithm 4, closely resembles the one described in Algorithm 3. The key distinction arises in Line 7 of Algorithm 4, as the choice of which comparator has to be trained depends on the average number of nodes of the R_{tot} sampled graphs. The comparator that is trained in that specific super-epoch is the one for which the average node value falls in the comparator's node interval. Moreover, the recursive tree and roll-out steps of 4 are performed employing the ensemble algorithm and both the initial graph distribution $\mathcal{D}_{\text{train}}^{\text{sorted}}$ and the dataset $\mathcal{S}_{\text{train}}^{\text{sorted}}$ are sorted in terms of the number of nodes.

Algorithm 4 Training Pipeline for the Ensemble Algorithm

Input: A distribution $\mathcal{D}_{\text{train}}^{\text{sorted}}$ over graphs.
Initialize parameters $\boldsymbol{\theta}_0^{(i)} \in \Theta$ for every comparator $\text{CMP}_{\boldsymbol{\theta}^{(i)}}$.
Initialize $\mathcal{S}_{\text{train}}^{\text{sorted}} = \mathcal{D}_{\text{train}}^{\text{sorted}}$. The datasets are sorted in terms of number of nodes.
for each super-epoch $t^{\text{super}} = 0, \ldots, T^{\text{super}} - 1$ **do**
    Pick the first R_{tot} graphs $[G_{\text{init}}^{(1)} = (V_{\text{init}}^{(1)}, E_{\text{init}}^{(1)}), \ldots, G_{\text{init}}^{(R_{\text{tot}})} = (V_{\text{init}}^{(R_{\text{tot}})}, E_{\text{init}}^{(R_{\text{tot}})})] = \mathcal{S}_{\text{train}}^{\text{sorted}}[0 : R_{\text{tot}}]$.
    Update $\mathcal{S}_{\text{train}}^{\text{sorted}}$ as $\mathcal{S}_{\text{train}}^{\text{sorted}} = \mathcal{S}_{\text{train}}^{\text{sorted}} - [G_{\text{init}}^{(1)}, G_{\text{init}}^{(2)}, \ldots, G_{\text{init}}^{(i)}, \ldots, G_{\text{init}}^{(R_{\text{tot}})}]$
    Choose $\text{CMP}_{\boldsymbol{\theta}^{(i)}}$ according to $N_{\text{avg}} = (|V_{\text{init}}^{(1)}| + |V_{\text{init}}^{(2)}| + \cdots + |V_{\text{init}}^{(R_{\text{tot}})}|)/R_{\text{tot}}$
    for each graph $G_{\text{init}}^{(i)}$ **do**
        Recursive tree: Run the ensemble algorithm and gather the leaf graphs
        Roll-outs: Estimate the MIS/MVC of each leaf graph
        Pairwise samples: Construct dataset by coupling each leaf node
    end for
    for each sub-epoch $t = 0, \ldots, T^{\text{sub}} - 1$ **do**
        Update the parameters $\boldsymbol{\theta}_{t+1}^{(i)} \in \Theta$ as in Equation 5.5 or 5.6
    end for
end for

Chapter 7

Experimental results

The objective of this chapter is to present both the experimental findings and the complete set of datasets and specifications utilized to achieve these results. Section 7.1 illustrates the employed datasets, including a special dataset artificially created by this work, while Section 7.2 lists the values of the most important hyper-parameters and the baselines used for comparing the results of this work with the ones obtained by other papers and methods. The most important result, namely the test set approximation ratio, is shown in Section 7.3, and Section 7.4 empirically tests the theoretical result of Sections 5.1.1,5.1.2 concerning the consistency property. Finally, an ablation study is conducted in Section 7.5.

7.1 Dataset specifications

The datasets over which the comparator is trained and tested are of two types. The first type, depicted in Section 7.1.1, is composed of classic datasets employed in diverse papers and works that deal with solving graph theory problems with the help of ML models. The second type, described in Section 7.1.2, is composed of an artificially built dataset. This dataset was built to show that the greedy heuristics, later described in Section 7.2.2, perform badly on specific graph distributions.

Two important parameters are considered for the analysis of the datasets: the size and the density of the graphs. The first parameter is measured in terms of the total number of nodes (namely $|V|$). The second one reflects how sparse a graph is and it is measured as the ratio between the total number of edges of the graph and the total number of edges if the graph had been complete: $\frac{|E|}{|V|(|V|-1)/2}$.

7.1.1 Real-world datasets and RB

The IMDB and COLLAB datasets [44] come from real-world scenarios. The IMDB is a movie collaboration dataset that gathers actor/actress and genre information from various movies on IMDB. COLLAB is a scientific collaboration dataset created by merging three public datasets focused on significant physics research problems. Both datasets contain *ego-graphs*, which are graphs having one node denoted as *hub* that is connected to all other nodes. As shown by Table 7.1, the two datasets differ in their sizes.

TWITTER dataset [45] has been created to model social-network scenarios. The graphs of TWITTER are obtained from ego-graphs by removing the hub node. The dataset is not particularly dense and contains, on average, graphs of more than 100 nodes (see Table 7.1).

The RB, RB200, and RB500 datasets have been purposely created by [46] to contain "hard satisfiable instances". Indeed, these datasets are known to be extremely challenging. The RB dataset is employed for the MIS problem while the RB200 and RB500 datasets, which differ in the number of nodes (see Table 7.1), are employed for the MVC problem. Notably, RB200 and RB500 were generated by configuring a small hyper-parameter $\rho = 0.25$, making the instances within these datasets challenging, as specified by [18].

Table 7.1: Statistics of the employed datasets. For each dataset, it is reported the average number of nodes and density of the graphs composing the dataset, and the total number of graphs employed for both training and testing (denoted as "Train" and "Test" respectively).

	IMDB	COLLAB	TWITTER	RB	SPECIAL (MIS)	RB200	RB500	SPECIAL (MVC)
Nodes	19.77	74.49	131.76	216.67	106.89	197.28	540.79	230.68
Density	0.51	0.52	0.205	0.218	0.530	0.205	0.179	0.527
Train $\lvert\mathcal{D}_{\text{train}}\rvert$	800	800	777	400	800	400	400	400
Test $\lvert\mathcal{D}_{\text{test}}\rvert$	200	200	196	100	200	100	100	100

7.1.2 Special dataset

As later shown in Tables 7.3,7.4, the greedy algorithms perform well on most datasets. So, why should somebody implement algorithms that leverage ML models? The answer to this question is given by the numbers of column SPECIAL of Tables 7.3,7.4, as the greedy heuristics do not perform well over the "special" dataset. Indeed, this dataset was purposely built by this work in order to show how easy it is to produce graph instances where greedy algorithms do not exhibit good

results. Indeed, the biggest advantage of ML-based algorithms compared to greedy heuristics is the generalization to every graph distribution, as ML-based solutions can be adapted to any kind of dataset, differently from greedy approaches.

The special datasets for MIS and for MVC share the same configuration, which is described in the next lines. The difference in the datasets lies in the average number of nodes of the graphs composing the datasets, as the special instances for MVC have bigger sizes than the ones for MIS (Table 7.1).

Whenever a new concept has to be described, it is not recommended to directly start from an example. In the case of the graph composing the special dataset, it might be better to start, instead, from the example of Figure 7.1. The figure shows a graph whose nodes are drawn in three colors. The nodes in blue, specifically b_1 and b_2, are called *boundary nodes* (B symbol) and are connected to only the nodes in red. The red nodes, that compose the set I, are denoted as *independent nodes*, and every node is connected to all nodes belonging to the B set and to all nodes belonging to the C set. It is important to remark that the nodes in I do **not** neighbor one another. Finally, the nodes belonging to the C set form a clique, and each node c_i (called *clique node*) is also connected to every node in I. In the example of Figure 7.1, the MIS coincides with the nodes in I while the MVC is the set of nodes C and B.

The special dataset is composed of graphs sharing the same configuration as the graph of Figure 7.1 but having a diverse number of sets B, I, C, namely $[B_1, \ldots, B_m], [I_1, \ldots, I_n], [C_1, \ldots, C_l]$. Moreover, the graphs for the special dataset were obtained by varying the number of nodes in every independent and clique set. If the number of nodes in every clique set is greater or equal to the number of nodes of every independent set, the MIS of each graph instance coincides with the independent sets, namely MIS $= I_1 \cup I_2 \cup \cdots \cup I_n$, while the MVC coincides with the union of the clique and boundary sets, namely MVC $= B_1 \cup B_2 \cup \cdots \cup B_m \cup C_1 \cup C_2 \cup \cdots \cup C_l$.

7.2 Training set-up and baselines

This section is dedicated to exploring the ensemble of hyper-parameters and methodologies used in the generation and evaluation of results for the proposed algorithms. It delves into the specifications of parameter settings and the techniques employed for both producing and assessing the outcomes of the proposed method. The analysis of the parameters set-up is in Section 7.2.1, while the description of the baseline methodologies is in Section 7.2.2.

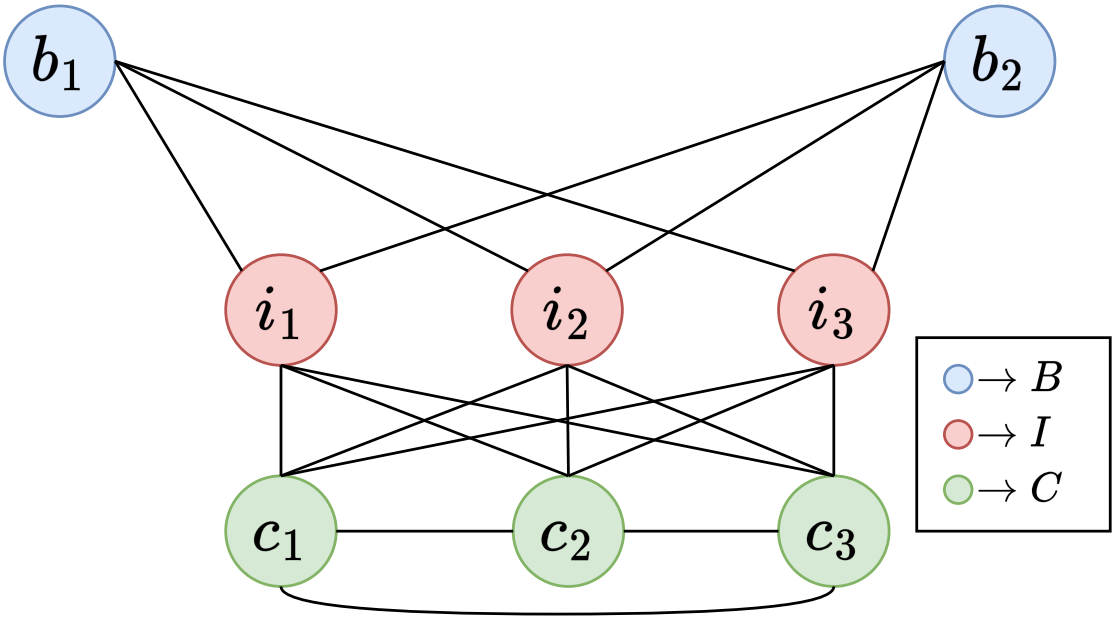

Figure 7.1: The image depicted in the figure is a specific example of a graph taken from the special dataset. The nodes of the graph are divided into three sets: boundary set (B letter), independent set (I letter), and clique set (C letter).

7.2.1 Parameters Set-up

As mentioned in Section 4.1, each macro-layer MacL_k is composed of three linear layers, and each of them captures a different semantic meaning of a node. Each linear layer has sizes 96×32 and the outputs of the three linear layers are concatenated and given as input to the linear layers of the next GEM iteration. As shown in Table 7.2, the GEM module has, in total, $K = 3$ iterations and, therefore, it contains 3 macro-layers.
The FCNN module is composed of $L = 4$ layers having the following sizes: 96×32, 32×32, 32×32, and 64×1. Furthermore, the output of the last GEM macro-layer has a skip connection [47] into the last dense layer of the model. The skip connection directly copies the output of the last GEM macro-layer and concatenates it with the output of the third FCNN layer, which is also why the last FCNN layer has 64 input neurons.
Regarding the training process, the model is trained with a learning rate of 0.001 and the Adam optimizer [48]. The datasamples described in Section 5.2.3 are gathered into batches of size equal to 32. The number of super-epochs and sub-epochs is respectively set to 10 and 30. Finally, the number of roll-outs M_{tot} defined in Section 5.2.2 is 10, and the number of picked graphs R_{tot} is 80, besides for the RB datasets and the special dataset for MVC where R_{tot} is set to 40.

Table 7.2: Experimental setting employed in the training process. The number of picked graphs R_{tot} is set to 40 for the RB datasets and the special dataset for MVC, while it is set to 40 for the remaining datasets.

Name	Value
Learning rate (λ)	0.001
Optimizer	Adam
Output layer size (D)	32
Number of GEM layers (K)	3
Number of fully connected layers (L)	4
Batch size	32
super-epochs (T^{super})	10
sub-epochs (T^{sub})	30
Number of roll-outs (M_{tot})	10
Number of picked graphs (R_{tot})	80, 40

7.2.2 Baselines

The results of various studies have been used as baselines to compare the outcomes of the proposed method. The paragraph *Baselines from related studies* makes a summary of them. Furthermore, a few greedy heuristics, illustrated in the paragraph *Baselines from heuristic methods*, are also employed as a baseline to compare the results of the proposed method.

Baselines from related studies: The work "RUN-CSP" [15] leverages a recurrent neural network and assigns a bi-dimensional state to every node, similarly to the MPNN structure described in Section 2.5.3. Moreover, a deep reinforcement learning approach is employed in the work "LwDMIS" [16]. In particular, the work of [16] uses an agent that iteratively determines either the belonging of a node to the final solution or defers the decision hoping that the belonging assignment will be easier in the next iteration. Finally, the approaches of "EGN" [17] and "Meta-EGN" [18] leverage an unsupervised learning approach (Section 2.3). Specifically, in the work of [17] an ML model is trained in order to minimize a relaxed version of the cost functions defined in Section 2.1.3. The work of [18] improves the theoretical studies of [17] by adding a smarter initialization to the model parameters.

Baselines from heuristic methods: The greedy algorithm for MIS iteratively chooses as part of the final solution the node v with the lowest degree of a graph, and removes all nodes neighboring v. The algorithm stops whenever the graph is composed of only isolated nodes, and the solution coincides with the total number

of isolated nodes.
The greedy algorithm for MVC operates using a similar principle. Firstly, all isolated nodes are removed from the initial graph. Secondly, several iterations are run and for each of them, the node u with the highest degree is part of the final solution. Similarly to the MIS case, all nodes neighboring v are taken away. Thirdly, whenever the graph consists solely of individual nodes, the solution aligns with the total number of these isolated nodes.
In addition to the results of the greedy algorithms for MIS and MVC, the outcomes of a *Simple Local Search* are taken into consideration. Given a time limit, the local search method randomly adds a node to the final solution and removes conflicting nodes. The largest or smallest (depending on the type of problem) set in the time period is used as the solution of the algorithm.
Finally, even the results of the proposed method obtained by a randomly parameterized comparator, denoted as *Random* CMP, are employed as a baseline.

7.3 Test Set Approximation Ratio Results

The most important criterion used to judge the performances of the proposed method and the methodologies illustrated in Section 7.2.2 is the "test set approximation ratio". Given one of the datasets of Section 7.1 and either the proposed algorithm for MIS or MVC, for every test graph from $|\mathcal{D}_{\text{test}}|$ a couple of values are evaluated. The first value is the solution to the problem (either MIS or MVC) given by the methods of Section 7.2.2 or by the proposed method. The second value is the optimal solution of either the MIS or MVC problem. This value is obtained by setting, for every graph, a time limit of 1 hour for the optimal solver Gurobi 10.0 [49]. Then, the division of the method's solution and the optimal value is evaluated. The final test set approximation ratio is the average value of such division among all graphs belonging to the test set $|\mathcal{D}_{\text{test}}|$ of a specific dataset. Naturally, the ratio is always lower or equal to 1 for the MIS problem, as the method's solution is always lower (or equal) than the optimal value. Conversely, the ratio for MVC is always greater or equal to 1.
It is important to remark that all results from the same column of Tables 7.3 to 7.6 are obtained using the exact same test dataset.

The results generated by the proposed approach can be categorized into two types. The initial type, as illustrated in Section 7.3.1, consists of outcomes obtained using the standard method, without the alterations described in Chapter 6. Conversely, the second category of results (presented in Section 7.3.2) is achieved through the modifications outlined in Chapter 6.

7.3.1 Standard method results

The descriptions of the results pertaining to the standard comparator and the baseline methods for both the MIS and MVC problems can be found in the following two paragraphs, presented separately.

Results for MIS: The data presented in Table 7.3 represent the test set approximation ratio for the MIS problem. The results for the RUN-CPS, EGN, and Meta-EGN methods are sourced from the researches of [17] and [18]. Notably, the results for the SPECIAL dataset are acquired by executing the open-source code referenced in the same research papers on the SPECIAL dataset. Additionally, all outcomes for the LwDMIS method across various datasets are obtained by utilizing the open-source code provided in the corresponding research paper.
Most methods perform optimally over the IMDB dataset, which is the smallest dataset, but struggle over the RB dataset, which contains the biggest graphs. Among the ML-based approaches, the proposed method (without modifications) achieves the best performance over the COLLAB and SPECIAL datasets. The Meta-EGN beats the standard comparator over the RB and TWITTER datasets. Remarkably, Table 7.3 shows a horrible result for the greedy MIS heuristic over the SPECIAL dataset (Section 7.1.2), as it performs even worse than the random comparator. Indeed, the heuristic always chooses the boundary nodes from $B_1, \ldots, B_m$, as they always have the lowest degree. Thus, all nodes from $I_1, \ldots, I_n$ cannot be chosen as they are not independent of the boundary nodes. Then, the greedy algorithm for MIS chooses one node from every clique set $C_1, \ldots, C_l$. Thus, the solution found by the greedy MIS always coincides with all boundary sets plus one node per clique set, even if the MIS is the union of all independent sets.

Results for MVC: The test set approximation ratios for MVC of Table 7.4 are taken from the work of [18], besides the results for the SPECIAL dataset obtained by running the open-source code of the related works. Similarly to the MIS problem, almost all methods perform optimally over the IMDB dataset. Moreover, regarding the ML-based approaches, the Meta-EGN method achieves the best performances across all datasets except for the RB500 and SPECIAL datasets, where the standard comparator has the best performance. As well as the MIS case, the modifications of Chapter 6 generally improve the results of the standard method, and the results of the Meta-EGN work may be beaten.
The greedy MVC algorithm does not behave well over the SPECIAL dataset. Indeed, the greedy heuristic chooses some of the independent nodes from $I_1, \ldots, I_n$, even if they are not part of the final MVC. Moreover, the boundary nodes are never considered as part of the final solution by the greedy algorithm, since their degree is lower than the one of the independent nodes I.

Table 7.3: Test set approximation ratios for MIS (the higher the better) on five datasets. The average approximation ratios (along with the standard deviation) on MIS are reported for the proposed method and the baseline methods described in Section 7.2.2. The results for the comparator are obtained from the standard method without the modifications of Chapter 6. For every dataset, the highest ratio among the ML-based baselines is reported in bold.

Method (↓) Dataset (→)	IMDB	COLLAB	TWITTER	RB	SPECIAL
CMP (standard)	**1.000**	**0.990 ± 0.051**	0.967 ± 0.083	0.770 ± 0.107	**0.996 ± 0.029**
RUN-CSP	0.823 ± 0.191	0.912 ± 0.188	0.909 ± 0.145	0.738 ± 0.067	0.946 ± 0.059
LwDMIS	**1.000**	0.978 ± 0.031	0.972 ± 0.032	0.804 ± 0.089	0.828 ± 0.304
EGN	**1.000**	0.982 ± 0.063	0.924 ± 0.133	0.788 ± 0.065	0.921 ± 0.218
Meta-EGN	**1.000**	0.988 ± 0.059	**0.976 ± 0.048**	**0.806 ± 0.059**	0.920 ± 0.228
Greedy MIS	1.000	0.998 ± 0.023	0.964 ± 0.048	0.925 ± 0.053	0.131 ± 0.055
Random CMP	0.874 ± 0.261	0.817 ± 0.211	0.634 ± 0.182	0.615 ± 0.155	0.225 ± 0.279
Simple local Search (10s)	1.000	0.860 ± 0.213	0.644 ± 0.218	0.565 ± 0.237	0.188 ± 0.340

Table 7.4: Test set approximation ratios (the lower the better) for MVC on six datasets. The average approximation ratios (along with the standard deviation) on MVC are reported for the proposed method and the baseline methods (Section 7.2.2). The results for the comparator are obtained from the standard method without the modifications of Chapter 6. For every dataset, the lowest ratio among the ML-based baselines is reported in bold.

Method (↓) Dataset (→)	IMDB	COLLAB	TWITTER	RB200	RB500	SPECIAL
CMP (standard)	**1.000**	1.011 ± 0.027	1.083 ± 0.044	1.031 ± 0.006	**1.015 ± 0.004**	**1.002 ± 0.001**
RUN-CSP	1.188 ± 0.178	1.208 ± 0.198	1.180 ± 0.435	1.124 ± 0.001	1.062 ± 0.005	1.051 ± 0.019
EGN	**1.000**	1.013 ± 0.022	1.033 ± 0.023	1.031 ± 0.004	1.021 ± 0.002	1.059 ± 0.025
Meta-EGN	**1.000**	**1.003 ± 0.010**	**1.019 ± 0.017**	**1.028 ± 0.005**	1.016 ± 0.002	1.063 ± 0.015
Greedy MVC	1.000	1.000 ± 0.004	1.015 ± 0.015	1.027 ± 0.007	1.014 ± 0.003	1.336 ± 0.278
Random CMP	1.012 ± 0.041	1.037 ± 0.050	1.149 ± 0.068	1.104 ± 0.019	1.086 ± 0.014	1.454 ± 0.318
Simple local Search (10s)	1.000	1.003 ± 0.011	1.078 ± 0.045	1.042 ± 0.005	1.024 ± 0.004	1.549 ± 0.093

7.3.2 Modified method results

The next paragraphs depict the results for the method with the modifications of Chapter 6, showing which modifications have the greatest impact.

Result for MIS: The first modification that has been implemented is the mixed

Table 7.5: The table presents the test set approximation ratio for MIS (where higher values indicate better performance) across five datasets. Each row corresponds to a distinct method. Specifically, the outcomes of the standard approach are included as well as those of five modifications described in Chapter 6, along with the top-performing ML-based baselines approach detailed in Table 7.3. The best result for each dataset is highlighted in bold.

Method (↓) Dataset (→)	IMDB	COLLAB	TWITTER	RB	SPECIAL
CMP (standard)	**1.000**	0.990 ± 0.051	0.967 ± 0.083	0.770 ± 0.107	0.996 ± 0.029
Mixed roll-outs	**1.000**	0.990 ± 0.049	0.977 ± 0.031	$\mathbf{0.836 \pm 0.083}$	0.994 ± 0.035
Concatenated model	**1.000**	0.990 ± 0.042	0.960 ± 0.046	0.767 ± 0.123	0.994 ± 0.039
Dataset augmentation	**1.000**	$\mathbf{0.996 \pm 0.025}$	0.972 ± 0.045	0.771 ± 0.109	0.996 ± 0.025
Layer Augmentation	**1.000**	0.996 ± 0.029	$\mathbf{0.977 \pm 0.030}$	0.773 ± 0.102	0.996 ± 0.026
Ensemble Learning	**1.000**	0.990 ± 0.044	0.977 ± 0.033	0.817 ± 0.086	$\mathbf{0.997 \pm 0.021}$
Best ML baseline	**1.000**	0.988 ± 0.059	0.976 ± 0.048	0.806 ± 0.059	0.920 ± 0.228

roll-outs one. As shown by Table 7.5, the results for MIS for such modification are better than the ones of the standard comparator, except for the SPECIAL dataset. This is possibly due to the bad behavior of the greedy heuristic when applied to the SPECIAL dataset. It is worth noting that since the mixed roll-out approach consistently performs better than the standard method across various datasets, the results for all other modifications in Table 7.5 were jointly obtained alongside the mixed roll-out modification. The dataset and layer augmentation adjustments show great results for the COLLAB dataset, but almost negligible improvements with respect to the standard method for the RB and SPECIAL datasets.
Unfortunately, the concatenated model does not exhibit great results. Capturing the interdependency of couples of graphs is in principle a good idea, but probably a deeper model is needed to model the complexity of such a relation.
Finally, the ensemble learning modification performs better than the standard comparator on every dataset, and it has the best result over the SPECIAL dataset. This modification leverages multi-comparators, so it is predictable that it performs better than the standard method that uses just one comparator.
Importantly, among the neural approaches, the modifications applied to the standard proposed method perform favorably in all datasets, as the ML-baseline results are exceeded. The performance of the method indicates that the proposed self-training scheme is able to learn from diverse data distributions and generalize reasonably well in the test sets of the respective dataset.

Table 7.6: The table displays the test set approximation ratios for MVC (where lower values signify superior performance) across six datasets. In particular, the table includes the results of the standard approach, alongside those of five modifications outlined in Chapter 6, as well as the top-performing ML-based baseline approach elaborated upon in Table 7.4. The lowest result for each dataset is emphasized in bold.

Method (↓) Dataset (→)	IMDB	COLLAB	TWITTER	RB200	RB500	SPECIAL
CMP (standard)	**1.000**	1.011 ± 0.027	1.083 ± 0.044	1.031 ± 0.006	1.015 ± 0.004	1.002 ± 0.001
Mixed roll-outs	**1.000**	1.008 ± 0.015	1.033 ± 0.028	1.027 ± 0.005	1.014 ± 0.004	1.005 ± 0.003
Concatenated model	**1.000**	1.045 ± 0.031	1.080 ± 0.041	1.035 ± 0.008	1.017 ± 0.006	1.003 ± 0.002
Dataset augmentation	**1.000**	1.008 ± 0.014	1.039 ± 0.029	1.027 ± 0.005	1.014 ± 0.005	1.002 ± 0.001
Layer Augmentation	**1.000**	1.005 ± 0.012	1.035 ± 0.028	1.029 ± 0.006	1.014 ± 0.004	1.002 ± 0.001
Ensemble Learning	**1.000**	1.009 ± 0.016	1.031 ± 0.024	$\mathbf{1.026 \pm 0.004}$	$\mathbf{1.013 \pm 0.003}$	$\mathbf{1.001 \pm 0.001}$
Best ML baseline	**1.000**	$\mathbf{1.003 \pm 0.010}$	$\mathbf{1.019 \pm 0.017}$	1.028 ± 0.005	1.016 ± 0.002	1.051 ± 0.019

Results for MVC: Similarly to the MIS problem, the mixed roll-outs modification shows better performances than the standard comparator in all datasets except for the SPECIAL one (see Table 7.6). Because of this, as well as the MIS case, all modification results are obtained with the mixed roll-outs adjustment as well.
Again, the complexity of the interdependency among graphs is not well-captured by the concatenated model, as it does not have better results than the standard comparator (except for the TWITTER dataset).
Finally, the ensemble learning approach has a big impact on the MVC problem, as shown by Table 7.6. Besides the COLLAB dataset, it achieves better performances among all modification approaches. Using multiple comparators implies that each comparator deals with a subset of all graphs from $|\mathcal{D}_{\text{test}}|$, and thus it can be more specific and precise.

7.4 Experimental analysis for the consistency property

What is stated in Theorems 3,4 is one of the cornerstone ideas of this work. Indeed, the goal is to train a comparator (either for MIS or MVC) in order to have it as consistent as possible, as a consistent comparator induces an algorithm that is optimal. Naturally, the comparators that produced the results in Section 7.3 are not consistent, as the test set approximation ratios are not equal to 1, with the exception of the IMDB dataset. Is there a way to check how far is a comparator from being consistent? The answer to this question is given in this section.

The consistency ratio of Figure 7.2 is a number between 0 and 1 measuring how far is a comparator from being consistent. The ratio is calculated during the training process. After every iteration, where an iteration is a total of 5 sub-epochs, the following is done. For every train graph from $|\mathcal{D}_{\text{train}}|$, the algorithm (either for MIS or MVC) is run for 10 times and the average value among the 10 runs is calculated. At this point, every train graph is associated with an average value. All possible pairs of graphs are formed, for a total of $\binom{|\mathcal{D}_{\text{train}}|}{2}$ couples. For each pair (G_0, G_1), the comparator is run to check if the comparator's choice is coherent with respect to the average values associated with the two graphs. As an example, if the consistency ratio for MIS is measured, a comparator for MIS is coherent if it chooses graph G_0 if and only if the average value of G_0 is greater than the one of G_1. If the comparator is coherent for a couple of graphs, then a "hit" is counted. At each iteration, the measured consistency ratio of the comparators of Figure 7.2 is the fraction of hits over the total number of graph pairs, namely $\frac{\#hits}{\binom{|\mathcal{D}_{\text{train}}|}{2}}$.

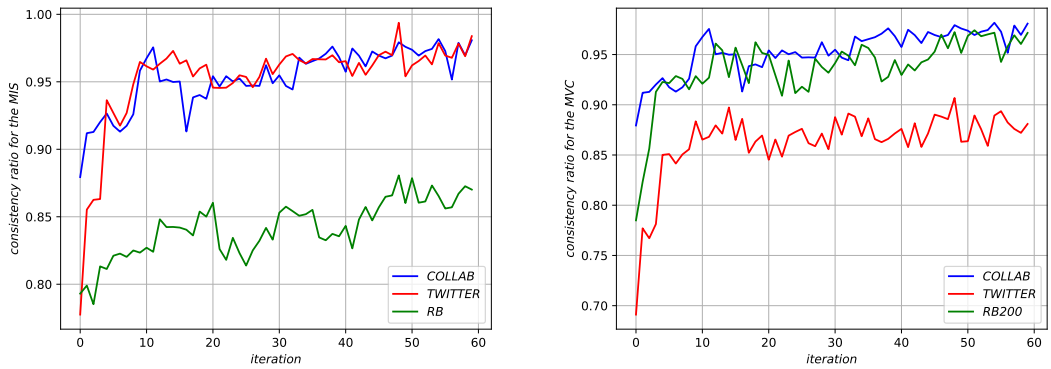

Figure 7.2: The left and right plots highlight the consistency ratio (the higher the better) of the comparator respectively for the MIS and MVC problems. For each plot, the consistency is calculated for three datasets and at the end of every iteration, where an iteration corresponds to 5 sub-epochs. The consistency ratio value associated with iteration 0 is obtained at the beginning of iteration 1 and indicates the consistency ratio of the random comparator.

The curves of Figure 7.2 have, overall, an increasing behavior for the 3 datasets reported in each plot. The 3 comparators in each plot become more and more consistent as training goes on. This experimental evidence is really important, as the more consistent a comparator the more it learns and can produce more accurate solutions. As a result, the two plots in Figure 7.2 are an experimental proof that the comparator, for both the MIS and the MVC problems, is learning after every

iteration.

7.5 Ablation study

This section proposes an ablation study on several parameters of the standard comparator function for MIS on the COLLAB dataset. The purpose of the study is to analyze how much the parameters under examination affect the final test results. The following parameters are tweaked:

- D: The output size of each of the 3 linear layers composing a macro-layer of the GEM module.
- K: The number of macro-layers (or iterations) composing the GEM module.
- L: The number of layers composing the FCNN.

The analysis of the 3 parameters is performed by changing them within the following ranges: $D \in$ [8,16,32,48,64], $K \in$ [1,2,3,4,5], and $L \in$ [2,3,4,5]. Moreover, the experiments are carried out with a base configuration of $D = 32$, $K = 3$, and $L = 4$. As an example, if the value $D = 64$ is tested, the parameters K and L are kept to their base configuration, namely $K = 3$ and $L = 4$.

Table 7.7: Test set approximation ratio for MIS and for the COLLAB dataset. The numbers are obtained by varying the parameter D. The highest value is reported in bold.

Parameter (D)	**8**	**16**	**32**	**48**	**64**
Model Performance	0.975 ± 0.058	0.970 ± 0.053	$\mathbf{0.990 \pm 0.051}$	0.983 ± 0.049	0.981 ± 0.050

Table 7.8: Test set approximation ratio for MIS and for the COLLAB dataset. The numbers are obtained by varying the parameter K. The highest value is reported in bold.

Parameter (K)	**1**	**2**	**3**	**4**	**5**
Model Performance	0.910 ± 0.100	0.946 ± 0.065	$\mathbf{0.990 \pm 0.051}$	0.989 ± 0.045	0.963 ± 0.058

The 3 parameters of Tables 7.7 to 7.9 reflect the complexity of the model $M_{\boldsymbol{\theta}}$ used to design the comparator $\text{CMP}_{\boldsymbol{\theta}}$. The higher the parameters value the higher the number of neurons composing the model $M_{\boldsymbol{\theta}}$. As shown by the 3 tables, if the

Table 7.9: Test set approximation ratio for MIS and for the COLLAB dataset. The numbers are obtained by varying the parameter L. The highest value is reported in bold.

Parameter (L)	**2**	**3**	**4**	**5**
Model Performance	0.920 ± 0.090	0.986 ± 0.057	$\mathbf{0.990 \pm 0.051}$	0.985 ± 0.056

values of the parameters are too low, the resulting model $M_{\boldsymbol{\theta}}$ does not have enough neurons, resulting in an under-fitting scenario. Similarly, a high parameter number leads to several neurons, and this can result in an over-fitting situation and poor performance.

Chapter 8

Conclusion and Future Work

In this work two novel DP-like algorithms, sharing the same structure, are proposed. The proposed methods are implemented to solve two classical graph theory problems, namely the Maximum Independent Set and Minimum Vertex Cover Problems. The work analyzes the usage of the comparator function, which is at the heart of the proposed approaches. Such a function is implemented with an ML model that is composed of a new module realized with graph neural networks, namely the GEM, and a standard fully connected neural network. Thanks to the comparator function, the algorithmic architecture of the proposed methods has been lightened, counteracting the space and time limitations that are typical of standard DP-like algorithms.

In this study, a self-supervised learning approach is employed to properly train the ML model. Self-training offers the dual benefits of data self-annotation and data generation, mitigating the lack of training data due to the NP-hardness nature of the MIS and MVC problems. The training process relies on the theoretical result denoted as consistency property, and the more a comparator is consistent the better the solutions that the induced algorithm can provide.

Remarkably, the proposed methods show good performance. Concerning the MIS problem, the ML-based baseline methods results are surpassed over all datasets. Regarding the MVC problem, the proposed approach is beaten only by the Meta-EGN work over just two datasets.

This work resulted in an article, which was accepted at the 37th NeurIPS conference[1].

8.1 Future work

Few adjustments could be implemented to improve the methods proposed in this work.

First of all, it could be helpful to employ another greedy heuristic for the mixed roll-outs modification of Section 6.1. Indeed, one could implement a randomized version of the greedy algorithms of Section 7.2.2, which randomly adds and removes nodes starting from the solution given by the greedy heuristic.

Secondly, it could be beneficial to enhance the ensemble learning modification capabilities. Indeed, this modification showed great performance, especially for the MVC problem. It could be possible to build artificial datasets resembling the ones of Section 7.1 but much easier to train. Therefore, the ensemble learning modification could be much more effective when trained over both the original dataset and the artificial one resembling it.

Thirdly, it could be extremely interesting to train only one model for all datasets. Up to now, each dataset required separate training and, therefore, several comparators. Thus, there is a one-to-one correspondence between each dataset and the corresponding trained comparator. What if one trains just one ML model over all datasets? Would the corresponding comparator generalize well across datasets having diverse graph distributions?

[1] *Maximum independent set: Self-training through dynamic programming.* Lorenzo Brusca, Lars C.P.M. Quaedvlieg, Stratis Skoulakis, Grigorios Chrysos and Volkan Cevher. 37th Conference on Neural Information Processing Systems (NeurIPS 2023)

Bibliography

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep residual learning for image recognition». In: 2016, pp. 770–778 (cit. on p. 1).

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is all you need». In: 2017, pp. 5998–6008 (cit. on p. 1).

[3] Zhihao Xing and Shikui Tu. «A graph neural network assisted monte carlo tree search approach to traveling salesman problem». In: *IEEE Access* 8 (2020), pp. 108418–108428 (cit. on p. 1).

[4] Yujiao Hu, Yuan Yao, and Wee Sun Lee. «A reinforcement learning approach for optimizing multiple traveling salesman problems over graphs». In: *Knowledge-Based Systems* 204 (2020), p. 106244 (cit. on p. 1).

[5] Marcelo Prates, Pedro HC Avelar, Henrique Lemos, Luis C Lamb, and Moshe Y Vardi. «Learning to solve np-complete problems: A graph neural network for decision tsp». In: vol. 33. 2019, pp. 4731–4738 (cit. on p. 1).

[6] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. «Learning to dispatch for job shop scheduling via deep reinforcement learning». In: 33 (2020), pp. 1621–1632 (cit. on p. 1).

[7] Junyoung Park, Sanjar Bakhtiyar, and Jinkyoo Park. *ScheduleNet: Learn to solve multi-agent scheduling problems with reinforcement learning*. 2021. arXiv: `2106.03051 [cs.LG]` (cit. on p. 1).

[8] Alex W. Nowak, Soledad Villar, Afonso S. Bandeira, and Joan Bruna. «A Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks». In: *ArXiv* abs/1706.07450 (2017) (cit. on p. 1).

[9] Gal Yehuda, Moshe Gabel, and Assaf Schuster. «It's not what machines can learn, it's what we cannot teach». In: *ICML*. PMLR. 2020, pp. 10831–10841 (cit. on p. 1).

[10] Wei Ou and Bao-Gang Sun. «A dynamic programming algorithm for vehicle routing problems». In: *2010 International Conference on Computational and Information Sciences*. IEEE. 2010, pp. 733–736 (cit. on p. 1).

[11] Kathrin Klamroth and Margaret M Wiecek. «Dynamic programming approaches to the multiple criteria knapsack problem». In: *Naval Research Logistics (NRL)* 47.1 (2000), pp. 57–76 (cit. on p. 1).

[12] Alane Marie de Lima and Renato Carmo. «Exact algorithms for the graph coloring problem». In: *Revista de Informática Teórica e Aplicada* 25.4 (2018), pp. 57–73 (cit. on p. 1).

[13] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. «A comprehensive survey on graph neural networks». In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24 (cit. on pp. 2, 13, 18).

[14] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. «Graph neural networks: A review of methods and applications». In: *AI open* 1 (2020), pp. 57–81 (cit. on p. 2).

[15] Jan Tönshoff, Martin Ritzert, Hinrikus Wolf, and Martin Grohe. «RUN-CSP: Unsupervised Learning of Message Passing Networks for Binary Constraint Satisfaction Problems». In: *CoRR* () (cit. on pp. 3, 58).

[16] Sungsoo Ahn, Younggyo Seo, and Jinwoo Shin. «Learning what to defer for maximum independent sets». In: *International Conference on Machine Learning*. PMLR. 2020, pp. 134–144 (cit. on pp. 3, 58).

[17] Nikolaos Karalias and Andreas Loukas. «Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs». In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6659–6672 (cit. on pp. 3, 58, 60).

[18] Haoyu Wang and Pan Li. «Unsupervised Learning for Combinatorial Optimization Needs Meta-Learning». In: *arXiv preprint arXiv:2301.03116* (2023) (cit. on pp. 3, 55, 58, 60).

[19] Irvin J Lustig, Roy E Marsten, and David F Shanno. «Interior point methods for linear programming: Computational state of the art». In: *ORSA Journal on Computing* 6.1 (1994), pp. 1–14 (cit. on p. 7).

[20] Eugene L Lawler and David E Wood. «Branch-and-bound methods: A survey». In: *Operations research* 14.4 (1966), pp. 699–719 (cit. on p. 7).

[21] James E Kelley Jr. «The cutting-plane method for solving convex programs». In: *Journal of the society for Industrial and Applied Mathematics* 8.4 (1960), pp. 703–712 (cit. on p. 7).

[22] John A Nelder and Roger Mead. «A simplex method for function minimization». In: *The computer journal* 7.4 (1965), pp. 308–313 (cit. on p. 7).

[23] James Aspnes. «Notes on Linear Programming». In: *Dept. of Computer Science, Yale University* (2004) (cit. on pp. 8, 9).

[24] Martin Grötschel, László Lovász, and Alexander Schrijver. «The ellipsoid method and its consequences in combinatorial optimization». In: *Combinatorica* 1 (1981), pp. 169–197 (cit. on p. 8).

[25] Richard Bellman. «Dynamic programming». In: *Science* 153.3731 (1966), pp. 34–37 (cit. on p. 9).

[26] Cameron B Browne et al. «A survey of monte carlo tree search methods». In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43 (cit. on p. 10).

[27] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. «The graph neural network model». In: *IEEE transactions on neural netwhttps://youtube.com/orks* 20.1 (2008), pp. 61–80 (cit. on pp. 13, 16–18).

[28] Weiwei Jiang. «Graph-based deep learning for communication networks: A survey». In: *Computer Communications* 185 (2022), pp. 40–54 (cit. on p. 13).

[29] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. «Graph neural networks and their current applications in bioinformatics». In: *Frontiers in genetics* 12 (2021), p. 690049 (cit. on p. 13).

[30] Lingfei Wu, Yu Chen, Kai Shen, Xiaojie Guo, Hanning Gao, Shucheng Li, Jian Pei, Bo Long, et al. «Graph neural networks for natural language processing: A survey». In: *Foundations and Trends in Machine Learning* 16.2 (2023), pp. 119–328 (cit. on p. 13).

[31] Mohamed A Khamsi and William A Kirk. *An introduction to metric spaces and fixed point theory*. John Wiley & Sons, 2011 (cit. on p. 15).

[32] Michael JD Powell. «An efficient method for finding the minimum of a function of several variables without calculating derivatives». In: *The computer journal* 7.2 (1964), pp. 155–162 (cit. on p. 15).

[33] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. «A survey of convolutional neural networks: analysis, applications, and prospects». In: *IEEE transactions on neural networks and learning systems* (2021) (cit. on p. 19).

[34] Hao Zhu and Piotr Koniusz. «Simple spectral graph convolution». In: *International conference on learning representations*. 2020 (cit. on p. 19).

[35] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. «Spectral networks and locally connected networks on graphs». In: *arXiv preprint arXiv:1312.6203* (2013) (cit. on p. 19).

[36] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. «Convolutional neural networks on graphs with fast localized spectral filtering». In: *Advances in neural information processing systems* 29 (2016) (cit. on p. 20).

[37] Thomas N Kipf and Max Welling. «Semi-supervised classification with graph convolutional networks». In: *arXiv preprint arXiv:1609.02907* (2016) (cit. on p. 20).

[38] Alessio Micheli. «Neural network for graphs: A contextual constructive approach». In: *IEEE Transactions on Neural Networks* 20.3 (2009), pp. 498–511 (cit. on p. 21).

[39] James Atwood and Don Towsley. «Diffusion-convolutional neural networks». In: *Advances in neural information processing systems* 29 (2016) (cit. on p. 22).

[40] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. «Neural message passing for quantum chemistry». In: *International conference on machine learning*. PMLR. 2017, pp. 1263–1272 (cit. on p. 22).

[41] Dan Hendrycks and Kevin Gimpel. «Gaussian error linear units (gelus)». In: *arXiv preprint arXiv:1606.08415* (2016) (cit. on pp. 34, 35).

[42] David Krueger et al. «Zoneout: Regularizing rnns by randomly preserving hidden activations». In: *arXiv preprint arXiv:1606.01305* (2016) (cit. on p. 35).

[43] G.P. Zhang. «Neural networks for classification: a survey». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* (2000) (cit. on p. 35).

[44] Pinar Yanardag and SVN Vishwanathan. «Deep graph kernels». In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, pp. 1365–1374 (cit. on p. 55).

[45] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014 (cit. on p. 55).

[46] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. «Random constraint satisfaction: Easy generation of hard (satisfiable) instances». In: *Artificial intelligence* 171.8-9 (2007), pp. 514–534 (cit. on p. 55).

[47] Keyulu Xu, Mozhi Zhang, Stefanie Jegelka, and Kenji Kawaguchi. «Optimization of graph neural networks: Implicit acceleration by skip connections and more depth». In: *International Conference on Machine Learning*. PMLR. 2021, pp. 11592–11602 (cit. on p. 57).

[48] Diederik P Kingma and Jimmy Ba. «Adam: A method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 57).

[49] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2023. URL: `https://www.gurobi.com` (cit. on p. 59).