POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Graph neural networks for the MCS problem

Supervisors

Prof. Stefano QUER

Prof. Giovanni SQUILLERO

Andrea CALABRESE

Candidate

Thomas MADEO

April 2022

Summary

The Maximum Common Subgraph (MCS) problem consists in finding the largest graph that is simultaneously isomorphic to a subgraph of two given graphs. Finding the MCS has been proven to be NP-hard, however it has many practical applications such as malware detection, cloud computing, drug synthesis, etc. The state-of-the-art MCS solver is based on the branch and bound paradigm. Its core section includes a node degree-based heuristic for node selection. This thesis extends and modify the MCS solver with the objective of finding a large solution in a limited budget time. We use Graph Neural Networks (GNN) to learn node embeddings and derive a better heuristic for the node selection. Our experimental analysis shows a substantial improvement over the original method. We also provide a solver variant which uses heuristics to directly selects node pairs instead of single nodes. Finally, we build a custom neural network to learn which nodes are likely to be part of a big solution by using the solver results as a ground truth. In these cases, performances are still under investigation, but the methodologies look promising.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graphs are a very versatile and effective data structure. They can be used to represent relational structures in many domains, such as chemistry (e.g., molecules), computer networks, road maps, or citation networks. In general, since any kind of network is composed of entities connected together, they can be naturally represented by graphs. Differently, other domains might need more effort to be represented as graphs. For instance, Wikipedia can be represented by a graph in which each node is an article, and each link represents the connection between two articles (e.g., a hyperlink).

In this thesis, we aim to extend and modify McSplit, the current state-of-the-art algorithm, to solve the Maximum Common Subgraph (MCS) problem with newer upshots from the deep learning domain. We will use Graph Neural Networks (GNN) and node embeddings, using McSplit or parts of it as a backbone. McSplit is optimized to find the exact solution in the shortest time possible. Since the MCS problem is proven to be NP-hard, the algorithm does not scale well for bigger graphs, requiring an enormous amount of time to find the solution. Instead, we aim to find a good enough solution within a limited time budget.

Graph Neural Networks have recently gained popularity for their ability to model the dependencies between nodes in a graph. GNNs are a class of deep learning methods that show exceptional performances in making inferences on data described by graphs. In this chapter, we will introduce some formal definitions of graph theory (Section 1.1) and computational complexity (Section 1.2). In Chapter 2,

we will talk more extensively of the MCS problem and its applications. We will introduce McSplit and its more advanced version based on reinforcement learning, which together represents the state-of-the-art solvers in Chapter 3. Since we will be using Graph Neural Networks, chapter 4 will be entirely focused on these machine learning methods. We will cover both classical machine learning models that work on graphs and GNNs. In chapter 5, we propose many different approaches to improve the state-of-the-art solver in finding a better incumbent, i.e., reaching a big solution size in a prefixed amount of time. We are not interested in finding the exact solution. We present our results in chapter 6 and draw our conclusions in chapter 7.

1.1 Formal Definitions

In this section, we present some definitions from graph theory, the branch of mathematics concerned with graphs [1].

Definition 1 *A **graph** $G = (V_G, E_G)$ consists of two finite sets, V and E. The elements of V_G are called **vertices** or **nodes**, and the elements of E_G are called **edges**. An edge has one or two nodes associated with it, which are called its endpoints.*

An edge joins (or connects) two nodes together. These nodes are said to be **adjacent**. An edge can also join a node with itself, creating a **self-loop**.

Definition 2 *A node that is joined to a node v is said to be a **neighbor** of v. The **neighborhood** of a node v in a graph G, denoted N(v), is the set of all the neighbors of v. If node v is included in N(v), we call it a **closed neighborhood**. Otherwise, we call it an **open neighborhood**.*

Definition 3 *The **degree** of a given node v is given by the cardinality of its neighbor set N(v).*

This definition can be generalized to an arbitrary order k. For instance, the 2-order neighborhood of a node v is given by N(v) plus all the neighbors of all nodes included in N(v). We denote it as $N_2(v)$. If we add to it all the neighbors

of nodes in $N_2(v)$, we obtain $N_3(v)$ and so on. In this thesis, when not otherwise stated, we will consider each neighborhood to be closed.

Definition 4 *A **directed edge** is an edge on which one of the two endpoints is designed as the source and the other as the target.*

Definition 5 *A **directed graph** is a graph G where all the edges in E_G are directed. On the contrary, when all the edges in E_g are not directed, we call it an **undirected graph**.*

If not specified, we consider all the graphs in this thesis to be undirected.

Definition 6 *The **order** of a graph G is the cardinality of V_G. The **size** of G is the cardinality of E_G.*

Definition 7 *A **subgraph** of a graph G=(V_G, E_G) is a graph H=(V_H, E_H) whose nodes in V_H are also nodes in V_G and edges in E_H are also edges in E_G.*

Definition 8 *For a given graph G, the **subgraph induced on a node subset** U of V_G consists in a graph G(U) whose node-set is U and whose edge set consists of all edges in G that have both endpoints in U. Otherwise is called a **non-induced subgraph**.*

In this thesis, we will consider only induced subgraphs.

Definition 9 *A **k-hop neighborhood** of a node v in a given graph G is the induced subgraph of G on the node subset $N_k(v)$.*

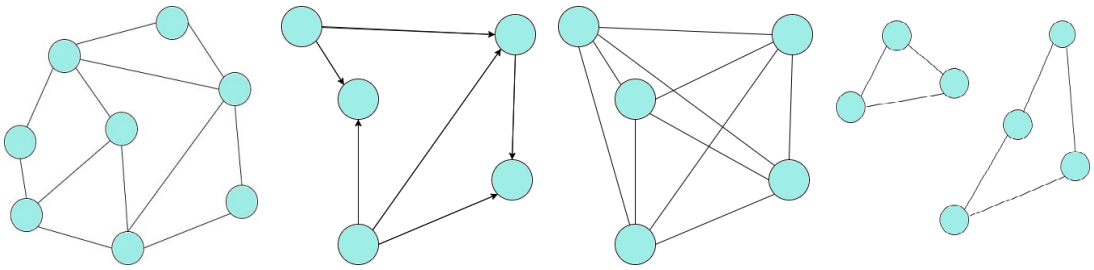
The main difference with the previous definition of a neighborhood is that in this case, we have a graph (i.e., both edges and nodes) instead of a simple set of nodes. We will extensively use this definition throughout the thesis.

Definition 10 *Two graphs G and H are **isomorphic** if there exists a node bijection $f : V_G -> V_H$ with the following property: For every pair of adjacent nodes u and v in G, the nodes f(u) and f(v) are adjacent in graph H. Similarly if u and v are not adjacent then f(u) and f(v) are not adjacent. Then if G and H are isomorphic, we denote it as $G \cong H$.*

Definition 11 *A **walk** is a sequence of nodes where every consecutive pair is adjacent. If there are no repeated nodes (except for the first and last nodes), we call it a **path**. If the starting node and the ending node in a path are the same, we call it a **cycle**. The length of the shortest path between two nodes is also referred to as their **distance**.*

Definition 12 *If there exists a path between every pair of nodes in a given graph G, we say G is **connected**. If G also has no cycles, it is called a **tree**.*

Definition 13 *A graph G is said to be **complete** if every pair of nodes is joined by an edge.*



(a) Undirected graph **(b)** Directed graph **(c)** Complete graph **(d)** Non connected

Figure 1.1: Examples of graphs

1.2 Computational Complexity

In this section, we will introduce the computational complexity theory. This will allow us to divide problems into different classes which have different properties [2]. Finally, we will find out to which class the MCS problem belongs.

Definition 14 *A **problem** is a set of instances together with a set of solutions for every instance. The set of solutions for a certain instance might also be empty. A problem **instance** is the input of a certain problem.*

Definition 15 *The **worst-case complexity** of an algorithm is the running time required given an input of arbitrary size. The **computational complexity** of an algorithm is usually its worst-case complexity.*

Definition 16 *We define the **time complexity** of an algorithm as a function that maps each instance to the running time of the algorithm needed to solve that problem instance. We denote it f(n), where n is the size of the problem instance.*

The computational complexity is usually denoted in asymptotic Big-O notation. We say that:

$$f(n) \in O(g(n)) \text{ if there exist } n_0, c > 0 \text{ so that } f(n) \leq c \cdot g(n), \forall n \geq n_0$$

In particular, if $g(n)$ is a polynomial, we say that the algorithm is of polynomial complexity. Otherwise, if the algorithm is $O(2^n)$ or $O(n!)$, we say the algorithm is of non-polynomial complexity. This is an important difference that will be used to divide problems into different classes. The computational complexity of a problem is defined by the complexity of the best algorithm capable of solving it.

Definition 17 *A **decision problem** is a problem that can be posed as a yes-no question of the input values.*

The different classes defined in the computational complexity theory all refer to decision problems. Any optimization problem can be expressed as a decision problem, usually by adding a bound.

Definition 18 *The class **P** is the set of all problems solvable by a polynomial algorithm.*

Usually, we refer to problems in P as easy since there are fast algorithms capable of solving them.

Definition 19 *The class **NP** is the set of all problems solvable by a non-deterministic polynomial algorithm.*

Definition 20 *A **non-deterministic polynomial algorithm** is an algorithm composed by two phases: We hypothesize to have an instance of a decision problem to which the answer is "yes". Then it can verify such a hypothesis in polynomial time.*

In other words, we don't have a polynomial algorithm capable of solving an NP problem. But if we are given a solution, there exists an algorithm that can verify it in polynomial time.

Definition 21 *A problem w is* ***reducible*** *to another problem p if for every instance of w we can build an instance of problem p in polynomial time. This implies that p is at least as difficult as w.*

It has been shown by **Cook's Theorem** that any problem in NP can be reduced in polynomial time to a particular problem called the **Satisfiability problem** (SAT). For this reason, the SAT problem is considered harder than the other NP problems.

Definition 22 *The class NP-complete is the set of all problems w that satisfies the following two conditions:*

- *w is an NP problem*
- *SAT can be reduced to w*

The NP-complete class represents the hardest problems in NP.

Definition 23 *A problem w is said* ***NP-hard*** *if there exists an NP-complete problem that reduces to it but, there is no proof that* $w \in NP$.

NP-hard problems are at least hard as the hardest NP problems. NP-complete problems belong both to NP and NP-hard. Usually the optimization version of any NP-complete decision problem is NP-hard.

Problems in NP, NP-complete, and NP-hard classes cannot be solved efficiently by exact algorithms. This means that we can only solve very small instances of these problems in a reasonable time. Usually, these problems are better handled by heuristics algorithms.

A very popular unsolved problem in computer science is to find out whether $P = NP$. There are no known algorithms that solve NP problems in polynomial time. However, from time to time, new algorithms are found, and problems that once were in NP are shown to belong to P. Since any problem in NP can be

reduced to an NP-complete problem in polynomial time, if an efficient algorithm is found for such problems, then it would be proven that all problems in NP are in fact solvable in polynomial time or that $P = NP$.

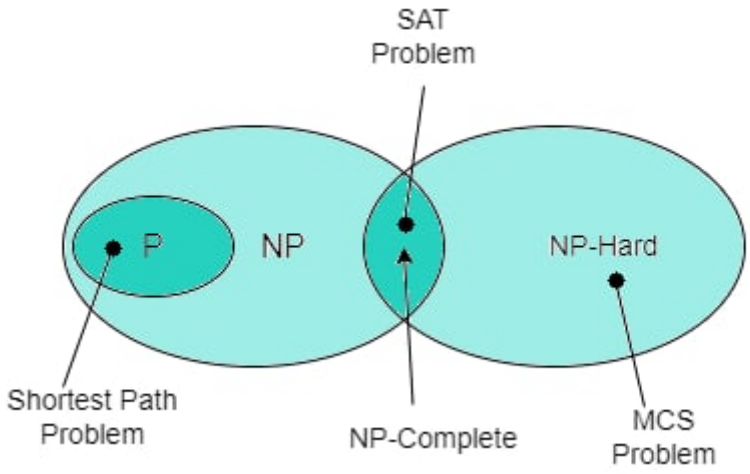


Figure 1.2: Graphical representation of the complexity classes

The Maximum Common Subgraph problem aims to find the biggest graph that is both a subgraph of two given graphs. It is proven that this problem is NP-hard. The decision version of the MCS problem can be easily defined by adding a bound, that is, finding a common subgraph with exactly n nodes. In this case, the problem belongs to the NP-complete class. It has also been proven that it is not possible to find an MCS approximation in polynomial time [3]. For this reason, most of the research is focused on finding efficient exact algorithms. In the next two sections, we will present some of the most prominent methods used to solve this problem and state-of-the-art algorithm known as McSplit.

Chapter 2

The MCS problem

In this chapter, we will talk about the Maximum Common Subgraph Problem. Section 2.1 will concentrate on the most famous methods used to solve the MCS, namely constraint programming and reduction to the maximum clique problem. In Section 2.2, we will talk about some real-world applications that would benefit from faster MCS solvers. The MCS problem has been defined in many ways. However, the two most popular formulations differ only for the maximization objective.

- The **Maximum Common induced Subgraph** of two graphs, G and H, is a graph that is an induced subgraph of both G and H and that has as many vertices as possible.
- The **Maximum Common edge Subgraph** of two graphs, G and H, is a graph that is isomorphic to both a subgraph of G and a subgraph of H with as many edges as possible.

Note that in the first formulation, we aim to maximize the number of nodes while in the second, the edges. In this thesis, we will only consider the induced version of the problem.

2.1 Known approaches

As stated in the previous section, the MCS problem is NP-hard either we use an exact algorithm or an approximation technique. All the methods presented are

therefore exact algorithms which aim to solve the problem as fast as possible.

2.1.1 Constraint Programming

Recent practical progress was obtained by using constraint programming [4] [5] [6]. It is based on backtracking and the use of smart branching and pruning strategies. The Branch and Bound (BnB) has been the most successful paradigm in approaching this problem. For the BnB, we need to define a "feasible solution" search tree. The root node contains all the solutions to the original problem. Each other node is the set of solutions to a simpler version of the problem (because we add some constraints). The leaf node corresponds to a single feasible solution. The BnB is divided into two phases:

- **Branching phase**: Starting from a node, we generate a certain number of children nodes according to a branching technique.
- **Bound computation:** We compute for each new node the upper(for a maximization problem) or lower(for a minimization problem) bound of the solution.

Of course, the method efficiency depends only on the bound computation. After the bound is computed at a certain iteration of the algorithm, each node of the tree with a lower (or higher) bound than the best solution value found until this iteration is closed; that is, it won't generate further children in the next branching phase. In this way, we obtain a powerful pruning technique.

2.1.2 Reduction to the Maximum Clique Problem

The Maximum Common Subgraph Problem can be reduced to the Maximum Clique Problem (MC). A clique is any subgraph of a given graph that is also complete. The MC problem aims to find a clique in a given graph that has the maximum number of nodes. Some methods propose to first apply the reduction and then solve the MC problem instead of the MCS [7] [5]. This strategy is shown to perform especially well on labeled graphs. We will not go into deeper detail as we will not use this kind of procedure.

2.2 Applications

The MCS problem has various practical applications. Finding the biggest common part between two graphs can be seen as a distance metric. In fact, the distance between two graphs is defined as the minimum number of operations (removing nodes or edges) necessary to make the two graphs isomorphic. In drug synthesis, a drug is a molecule that can be represented by a graph. Finding the maximum common part in two molecules can be helpful to predict how the human body reacts to certain drugs. Other applications include: Malware detection, analysis of electronic circuits, or computer vision.

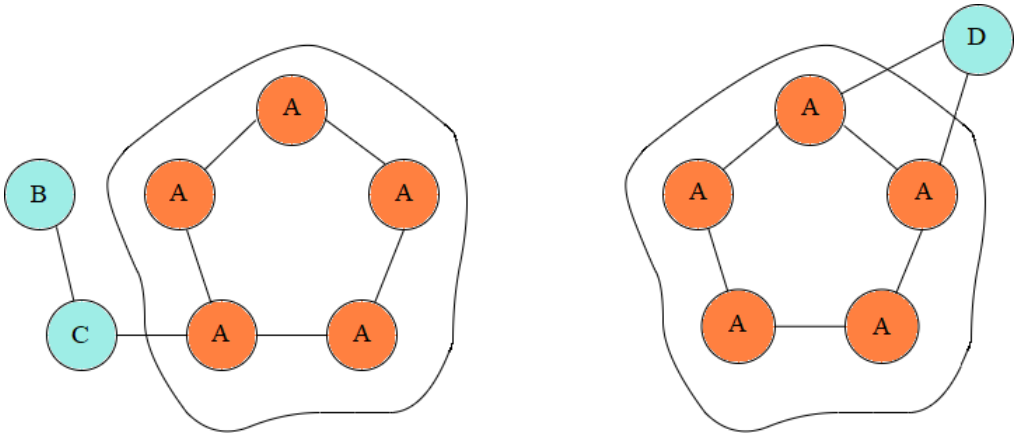


Figure 2.1: Example of an MCS instance solution

Chapter 3

State of the art

Since all the work in this thesis uses McSplit [6] as a backbone, in this chapter, we will present the algorithm in detail. After a quick overview in section 3.1 , we will introduce some new concepts on which McSplit extensively depends (Section 3.1) . This will be followed by a practical example execution (Section 3.2). Finally, we also present a newer McSplit variant that uses reinforcement learning (Section 3.3).

3.1 Overview

McSplit is a recursive branch and bound algorithm which exhaustively checks all possible matches between the nodes of two input graphs. Two graphs are given as an input, namely G and H, and the algorithm will start by selecting one node v from G. Node v is then matched with each node w from graph H. However, a new node pair is only considered if two conditions are met:

1. The nodes are connected in the same way to all previously selected nodes.
2. The new pair can potentially lead to a common subgraph larger in size than the incumbent solution.

These two conditions allow the algorithm to efficiently prune the search space. This is done with the intent of maximizing the pruning effect of the algorithm. The nodes selection, at each recursion level, is made by using two heuristics on which the algorithm performances significantly depend.

The algorithm is specifically designed to solve the **induced** version of the MCS problem. In the original paper, they show improvements by more than an order of magnitude on the unlabeled variant of the problem. Furthermore, due to lower memory usage in respect to previous constraint programming and clique approaches, it can also handle much larger instances of the problem.

Adjacency Class - Bidomain

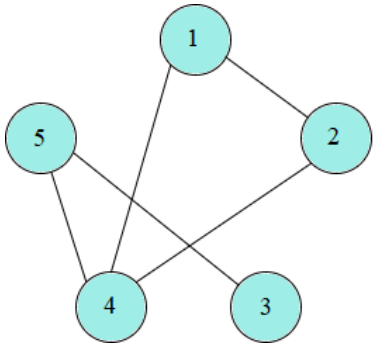


Figure 3.1: Example Graph

As we said, when branching, a new node pair is considered if and only if the nodes are connected in the same way to all previously selected nodes. That is, the nodes have the same connectivity pattern. This is ensured by re-labeling the nodes after a new node pair is selected. These labels are called **Adjacency Classes**.

For instance, let's consider the graph G in (Fig 3.1). Initially, all nodes are unlabeled; that is, they belong to no class. Let's suppose we are running McSplit, and as a first step, we select node 2 using some heuristic. This node will then be matched with all nodes in the other input graph, but in this case, we will focus only on the node re-labeling of G. The same behavior will happen around the select node in the other input graph. All nodes adjacent to node 2 will be labeled with 1. All nodes non-adjacent to node 2 will be labeled with 0. This way, label 1 means that a node is connected with the first select node while 0 means the opposite.

Now, let's select another node, for instance, 4. 4, which has a label of 1, will be matched with a node with the same label in the other input graph. In graph G,

all nodes except for 2 and 4 are re-labeled again by adding another binary digit to the previous label. In the end, after the selection of node 4, we will have four adjacency classes.

The set of nodes belonging to a certain adjacency class is also called a **domain**. While running McSplit, we will be relabeling nodes on both input graphs, and they will always share the same adjacency classes. However, the domain for each class can be different. The set of nodes belonging to the same adjacency class but considering both input graphs is called **bidomain**, which can be compactly stored in memory.

3.2 Example Execution

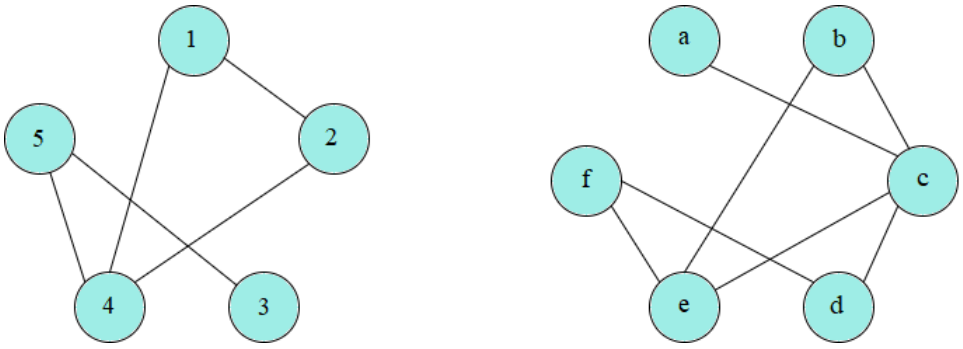


Figure 3.2: Sample graphs G and H

For this example, we will consider both input graphs to be un-labeled, undirected, and without self-loops (nodes connected to themselves). The input graphs we will consider are those in namely $G = (V_G, E_G)$ and $H = (V_H, E_H)$. Given $v_i \in V_G$ and $w_i \in V_H$, the objective of the algorithm is to find a mapping $M* = (v_1, w_1), ...(v_m, w_m)$ with the maximum-cardinality, that is with as most node pairs as possible. To ensure the isomorphism, taken two nodes, v_i and v_j, from the mapping, they must be adjacent in G if and only if their respective matches w_i and w_j are adjacent in H.
McSplit will first select a node in $v \in V_G$ using a maximum degree heuristic, and then it will pair v with all nodes $w_i \in V_H$, which are also ordered by node degree.

Of course only nodes with the same adjacency class can be paired together, but on the first iteration, all nodes belong to the same adjacency class. Since node 4 in G is the node with the maximum degree, it will be selected. Node 4 will now be matched with all nodes in H, starting from the maximum degree node, which is c. We now have $M = [4c]$. All remaining nodes will need to be re-labeled. If a node is adjacent to 4 or c, it will be labeled with 1, otherwise, it we be la-belled with 0. Now we have to select the second node in G. To select a second node, it is not sufficient to consider the node degree as we now have 2 different adjacency classes. A second heuristic will first choose an adjacency class, and then we will select the node with the maximum degree that belongs to the chosen class. McSplit will choose the adjacency class with the smallest $max(|G|, |H|)$, i.e., for each class, we count the number of nodes in G and H and select the max value. We then take the class with the smallest max value. To better follow our example, take a look at table 3.1. In our case:

- Class 1 has three nodes in G and four nodes in H. Therefore, its final score is 4.
- Class 0 has one node in G and one node in H. Therefore, its final score is 1.

Class 0 is selected, and since there is only one node in each class, the mapping comes quickly. The new mapping is $M = [4c, 3f]$. All nodes in both graphs must be re-labeled for a new class selection. The new situation is:

- Class 11 has one node in G and two nodes in H. The final score is 2.
- Class 10 has two nodes in G and two nodes in H. The final score is 2.

Ties are broken by selecting the adjacency class that contains the node with the maximum degree. In this case, that node is e which belongs to class 11. The node with maximum degree in G is paired with the maximum degree node in H, so the next pairing will be $5e$. Our current matching is $M = [4c, 3f, 5e]$. After all nodes have been re-labeled, we can notice that two classes are present in graph H but not in graph G. These nodes cannot be considered since there won't be any match in G for them. The last matching is $M = [4c, 3f, 5e, 1a]$. The incumbent solution until now is 4, which is the size of the mapping M. The algorithm is now forced to

backtrack, but from now on, only branches with a computed bound higher than four will be considered.

Table 3.1: Step by step node labeling for both graphs

Mapping	**Labeling of G**		**Labeling of H**	
	Node	**Label**	**Node**	**Label**
M=[4c]	1	1	a	1
	2	1	b	1
	5	1	d	1
	3	0	e	1
			f	0

Mapping	**Labeling of G**		**Labeling of H**	
	Node	**Label**	**Node**	**Label**
M=[4c, 3f]	1	10	a	10
	2	10	b	10
	5	11	d	11
			e	11

Mapping	**Labeling of G**		**Labeling of H**	
	Node	**Label**	**Node**	**Label**
M=[4c, 3f, 5e]	1	100	a	100
	2	100	b	101
			d	110

Mapping	**Labeling of G**		**Labeling of H**	
	Node	**Label**	**Node**	**Label**
M=[4c, 3f, 5e, 1a]	2	1001	b	1010
			d	1100

The bound is computed with the formula:

$$bound = |M| + \sum_{l \in L} min(|\{v \in V_{G,l}\}| \ , \ |\{v \in V_{H,l}\}|) \tag{3.1}$$

Here, the set of adjacency classes is denoted by L. As an example, let's suppose to be in the situation like in 3.1 at the third depth level. We have a matching $M = [4c, 3f, 5e]$ which is of size 3. Our incumbent, i.e., the best solution found until now, could be of a different size, let's call it $|M*|$ There are three classes, but among them, only class 100 can be considered since 101 and 110 are available only in one graph. Before matching nodes, we should first compute the bound for class 100. Class 100 is found twice in G and once H; therefore, only one pair can be matched in the best case. The final value for the bound is $M + 1 = 4$. Now, if our incumbent solution size $|M*|$ is 4 or higher, this matching is useless as it will not bring us an improvement. This branch is therefore closed, and the algorithm backtracks without wasting computational resources.

3.3 McSplit + Reinforcement Learning

Reinforcement learning is a particular machine learning method based on an agent that is rewarded when it acts in a desirable way and punished otherwise [8]. The agent interacts with the external world, it has a certain state and produces an action. The external world gives the agent feedback on the performed action. The reward or punishment is used to update the agent state and perform better actions in the future. Basically, the agent learns by trial and error, just like humans do.

This framework has been used to solve the MCS problem by adding an agent to the normal implementation of McSplit. This new method has been called McSplit+RL [9]. The reinforcement learning agent is used for learning to discover the best branching choices, i.e., the branches that reduce the search tree size. Each branching choice is considered an action; when the agent performs a branch, it receives a reward based on the reduction of the search space. At every branching point, the agent will select the branch with the highest expected reward. Given a label class, McSplit always branches on nodes with the highest degree. McSplit+RL can branch on nodes with a lower degree if the reinforcement learning agent expects

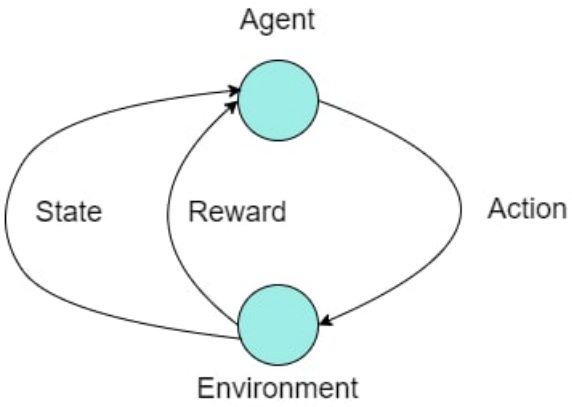


Figure 3.3: The typical framing of a Reinforcement Learning (RL) scenario: an agent takes actions in an environment, which is interpreted into a reward and a representation of the state, which are fed back into the agent.

a higher reward from them. Apart from easy instances, in which the reinforcement learning overhead is more computationally expensive than solving the problem itself, McSplit+RL has been shown to perform better than McSplit. Both methods want to find the optimal solution for the MCS. When tested on about 3000 problem instances, McSplit+RL has been able to solve about a hundred more of them before a prefixed 1800 seconds deadline.

Chapter 4

Graph Neural Networks

In this chapter, we will introduce the machine learning models we used to improve McSplit. First, we will talk about one of the simplest machine learning models: The perceptron. In function of it, we will resume the main machine learning concepts. We will then discuss the simplest neural network architecture, namely the multilayer perceptron (Section 4.1). Finally, we will talk about machine learning methods that are built specifically for graphs (Section 4.2) with emphasis on Graph Neural Networks (Section 4.3).

4.1 Multilayer perceptron and Machine Learning Fundamentals

The perceptron is a mathematical algorithm inspired by biological neurons. Neurons in our brain constantly receive signals. Each signal activates some synapses in the neuron. The activated synapses measure how important each input signal is. An output is then produced and carried by an axon. In a perceptron, the input is simply a vector of numbers, and the synapses are modeled by another set of numbers called **weights**. A perceptron is a binary classifier, i.e., given an input, it can decide whether it belongs or not to some specific class.

Usually, the input of a perceptron is a data point with each feature represented by a number. For example, if we were analyzing images with a 256x256 resolution, a single image can be represented by a vector of size 256x256, where each number

represents the color of the corresponding pixel. Each element of the input vector is then multiplied by a corresponding weight. All the results are then summed together with a bias and fed to a non-linear activation function to obtain the output.

The perceptron is a machine learning model with the objective of learning the correct weights. Suppose that each input belongs or not to a class; we will have two possible labels.

- 1: Represents an input instance belonging to that class
- 0: Represents an input instance not belonging to that class

After an input instance is processed by the perceptron, it will **predict** its label. We want to find a set of weights so, the number of mistakes on a set of general input instances is minimized.
In order to do so, before being deployed, a perceptron will undergo three phases:

- Training phase: In which the weights are tuned.
- Validation phase: In which the **hyperparameters** are tuned.
- Test phase: A final performance evaluation.

A dataset, i.e., a collection of sample input, is required to run the three phases. Each phase will use a different portion of the original dataset. Usually, most of the dataset is used in the training phase.
The training phase consists in feeding the perceptron (initialized with random values) with the training dataset. We will obtain a prediction for each input instance in the training set. We are also supposing to know already the correct prediction of each instance. Given that, we are able to compute a performance evaluation using a **loss function**. The loss function measures the discrepancy between our predicted labels and the expected ones. The objective of the training phase is to minimize the loss function score. This is done by another algorithm called **Gradient Descent**. At each step, the whole dataset is fed to the perceptron, and after the loss computation, the weights are updated in the steepest descent direction. The descend direction is found computing the function gradient. Usually, this is repeated until the loss function reaches a local minimum. One thing we can choose

when updating the weights is the step size, also called the **learning rate**. The learning rate is simply a number that multiplies the gradient value, and it is also a hyperparameter of the problem; it must be chosen before starting the training phase. To find the best learning rate as well as the best set of hyperparameters (usually, we have around 5 of them), we require the validation phase. The validation phase consists in choosing a prefixed range of acceptable values for each hyperparameter and then train the perceptron with all the different combinations of their values. Each trained perceptron is tested on a specific portion of the dataset called the validation set. The best perceptron is kept and goes to the testing phase. In the testing phase, we evaluate the perceptron on yet another portion of the dataset. It is important that the model has never seen these instances, otherwise, the results would be biased. In figure 4.1, we can see a graphical representation of a perceptron.

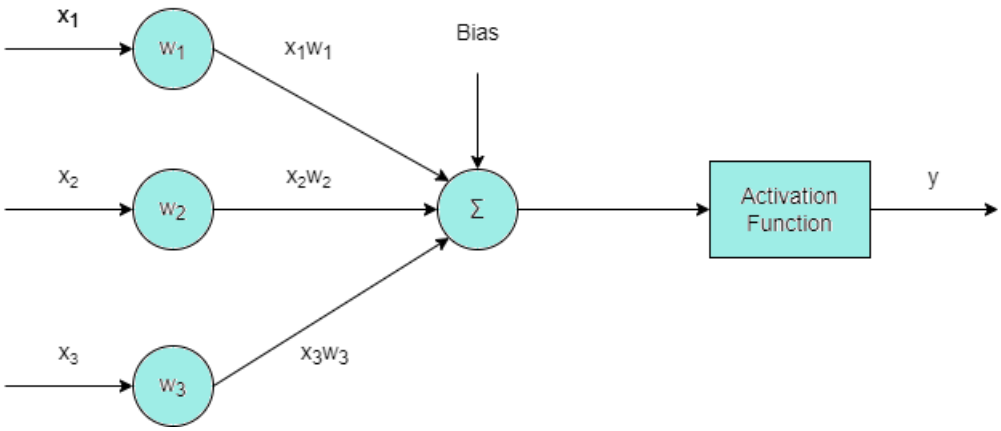


Figure 4.1: Perceptron scheme.

Multiple perceptrons can be stacked; the resulting architecture is the simplest form of neural network called **Multilayer Perceptron** (MLP) [10]. Each perceptron is called a layer. Each layer works with the output of the previous layer, except for the first one, which works directly on the input. Each layer has an output, and the last layer output is our final result. In order to have a deep neural network, we need at least three layers, i.e., one layer more than the input and the output layers. The middle layer is called a hidden layer, and we can have an arbitrary number of them. Of course, training a MLP is a much slower process than training a single

perceptron. Since after each perceptron we have a non-linear activation function, this model can also approximate non-linear functions. In figure 4.2, we can see a graphical representation of the MultiLayer perceptron.

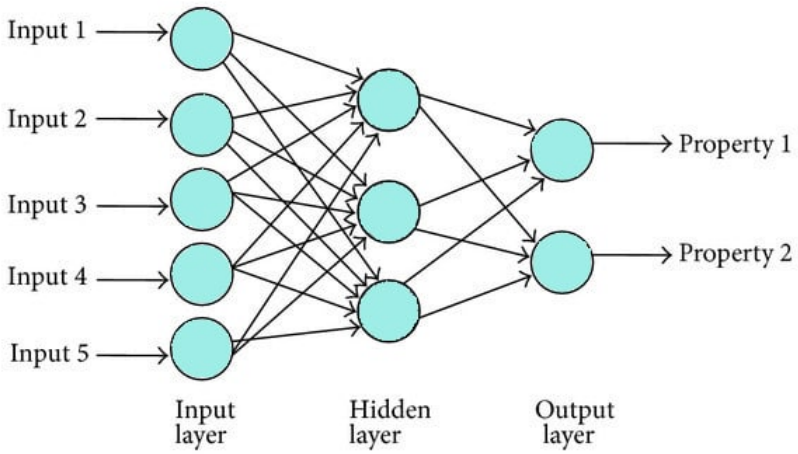


Figure 4.2: MultiLayer Perceptron

4.2 Machine Learning with Graphs

In this section, we will present the basics of applying machine learning to graphs. The most simple way to approach graphs is by feature engineering, that is, defining a feature vector "by hand". Then we will see some more evolved methods capable of learning a representation automatically. These methods go under the name of representation learning. Most importantly, we will introduce the WL-kernel and the concept of **node embedding**, which are closely related to Graph Neural Networks.

4.2.1 Feature Engineering Methods

The traditional Machine Learning pipeline is all about designing proper features. There are two kind of features: node's features (like proteins that have different properties) and structural features that describes how these nodes are placed in the

rest of the graph. By designing the correct features, we will be able to make better predictions. Therefore the first step is converting each graph to a vector of features. These vectors can be used to train any traditional classifier like **Support Vector Machines**, **Random Forests** and so on. Some common node level features are:

- The node degree
- Eigenvector centrality: The node importance is computed as the normalized sum of the importance of its neighbors
- Betweenness centrality: A node is important if it lies on many shortest paths between other pairs of nodes.
- Closeness centrality: A node is important if it has a small shortest path length to all other nodes.
- Clustering coefficient: Measures how connected a node neighbors are.
- Graphlets: Count pre-specified graphs in the neighborhood of a given node.

In particular, graphlets are rooted, connected, and non-isomorphic graphs (Fig 4.3). To describe a node using graphlets, we can build a graphlet degree vector: We choose a node and count all the possible graphlets that node participates in. Each cell of the vector will contain the final count for a specific graphlet. Graphlets are an example of a structural feature, while the node centrality measures are more like node features.
Sometime we might want features that characterize the structure of the entire graph. This will allow us to compute the similarity between two given graphs. Of course, the notion of similarity can vary. To solve this problem, usually **kernel methods** are used. A kernel is a function that measures similarity between two things (graphs), i.e., a function that takes two vectors as an input and gives a value between 0 and 1 as a result. More specifically, the kernel function represents the dot product between a specific feature representation of the two graphs. We will present the **Weisfeiler-Lehman Kernel** (WL) which is closely related to **Graph Neural Networks**.

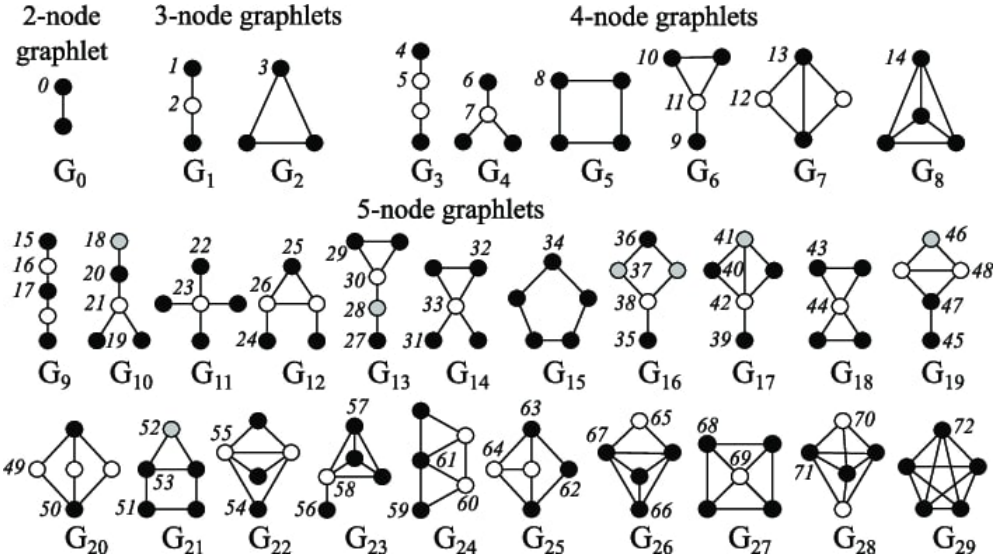


Figure 4.3: All possible graphlets with up to five nodes.

WL-kernel

The WL kernel [11] generalizes the concept of **bag of node degrees** . A bag of node degrees is simply a vector that counts how many nodes with a certain node degree value are present in a graph. For instance, a bag of node degrees $V = [2, 0, 4]$ means that the represented graph has two nodes with degree 1, 0 nodes with degree 2 and 4 nodes with degree 3. The idea of the WL kernel is to iteratively enrich the node vocabulary generalizing the node degree by collecting degree information from a k-hop neighborhood. The algorithm used to achieve this is called **Color Refinement**. Given a graph $G = (V_G, E_G)$ we initially assign a color $c^{(0)}(v)$ to each node $v \in V_G$. At iteration k, we refine node colors with the formula:

$$c^{(k+1)}(v) = HASH\left(\left\{c^{(k)}(v), \left\{c^{k}(u)_{u \in N(v)}\right\}\right\}\right) \tag{4.1}$$

The HASH function must be a bijective function, i.e., it maps different inputs to different colors. This way, at step K the color $c^{(K)}(v)$ summarizes the structure of the k-hop neighborhood of a node. After the last iteration, we compute a bag of color vector. Each element of the vector represents the total count of nodes with a certain color. We can then repeat the same procedure for a second graph

and compute the dot product between the two bag of color vectors to obtain a similarity measure. A graphical example can be found in Fig 4.4.

4.2.2 Representation Learning

With the use of representation learning methods, we aim to remove a tricky step from the machine learning pipeline: The need for feature engineering. We are going to learn a function that maps every node into a d-dimensional feature vector. Such vector is called an **embedding**. Nodes which are similar in the graph should be close also in the embedding space. Node not similar in the graph should be not similar in the embedding space. We need to define three things:

- A notion of similarity between nodes in the graph.
- An encoder: a function that maps nodes to the embedding space
- A decoder: a function that maps embeddings to a similarity score (usually the dot product).

The encoder parameters must be optimized so that the similarity function on the embedding space is as close as possible to the similarity function in the graph. Usually, an embedding vector dimension goes from 64 to 1000. Before discussing deep learning and graph neural networks, we would like to introduce a "shallow" embedding method called node2vec.

Node2Vec

The basic idea of node2vec [12] is to compute a matrix. A column of the matrix will represent a specific node embedding. We directly optimize the node embeddings (With GNNs we will optimize the **parameters** of the network like on the Multilayer Perceptron). Our objective is to find the optimal matrix, a set of embeddings such that the dot product between them represents our notion of similarity. Node2Vec is based on the concept of **Random Walk** to compute embeddings. A random walk is a walk generated with some random strategy. For example, an unbiased random walk might consider all neighbors nodes as the next step with equal probability. Random walks are flexible and incorporate both local and high order neighborhood

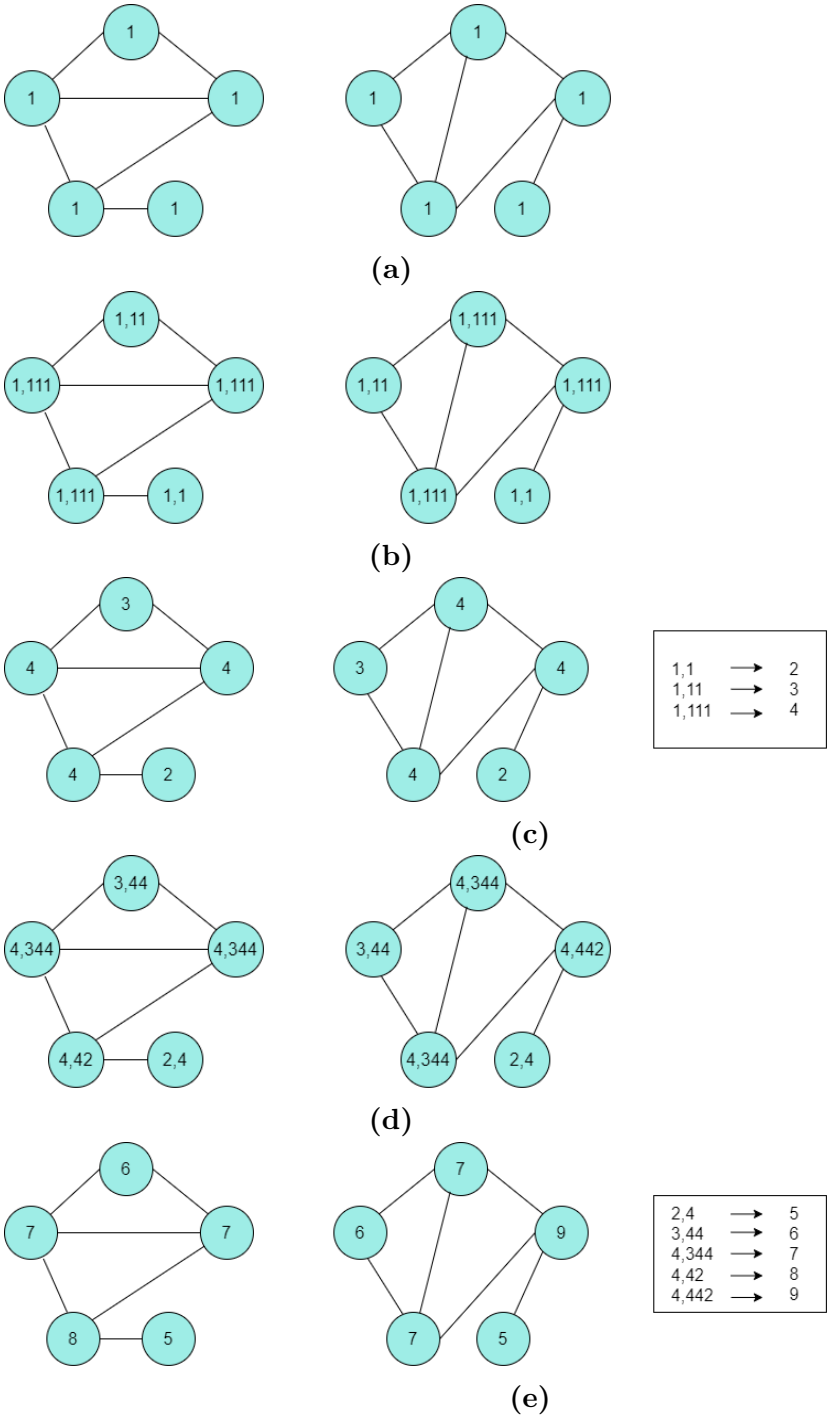


Figure 4.4: WL-Kernel application example

information. We are interested in node pairs that co-occur in a random walk. We want nodes that appear in the same random walk to be close together in the embedding space. Summarizing, the main steps to optimize the embedding space using random walks are:

- Running a fixed number of random walks starting from each node.
- For each node, we collect its neighborhood. The neighborhood, in this case, it is the multiset of the nodes visited using random walks.
- Optimize the embedding space according to: Given a node predict its neighborhood.

Given a graph $G = (V_G, E_G)$, The optimization problem in the third step can be represented by the following loss function:

$$\sum_{v \in V_G} \left(\sum_{u \in N_R(v)} \left(-log \left(P \left(u | z_v \right) \right) \right) \right) \tag{4.2}$$

Where $N_R(v)$ represents the neighborhood of node v obtained using a random walk strategy R and z_v is the embedding vector of node v. The probability $P(v|z_u)$ can be modeled using a softmax function:

$$P \left(u | z_v \right) = \frac{exp \left(z_v^T z_u \right)}{\sum_{n \in V_G} \left(exp \left(z_v^T z_n \right) \right)} \tag{4.3}$$

The softmax function simply normalizes a vector of real values into probabilities (values that sum up to 1). This optimization problem can be then solved by using the Gradient Descent algorithm. Since in this case, the number of random walks can be huge, usually the gradient descent is computed on batches of training samples. This algorithm is called **Stochastic Gradient Descent**. Now we only have to define a random walk strategy. **Deep Walk** [13] is an embedding method that simply uses unbiased random walks. Node2Vec generalizes this concept by adding two hyperparameters. An in-out parameter q is added in order to bias the random walks to be more prone on exploring the graph in depth (as with a Depth First Search) or by breadth (as with a Breadth First Search). A return parameter p is

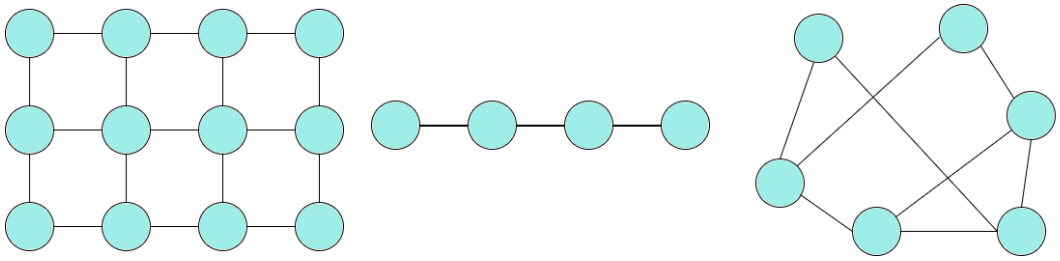
added to control the probability of returning to the previously visited node in the random walk.

4.3 GNN

The Multilayer Perceptron is a very general deep learning framework. It works with virtually any kind of data structure that can be represented as a vector of some sort. In deep learning, sometimes we need special purpose architectures that exploit some unique characteristics of a data structure to make much better predictions with less training time. A first example of such deep neural networks are Convolutional Neural Networks (CNN). CNNs are optimized to solve tasks on images. While images can be represented as a vector, this way, we do not exploit well the spatial information the image itself carries. Moreover, images tend to have repeating patterns (e.g, the windows of a building are all the same) and for this reason it is convenient to share neuron's weights. The weight sharing between neurons is by performing the multiplication between the input and a sliding weight matrix. Another example are Recurrent Neural Networks (RNN), which are optimized for sequences. The most important use case is text analysis. As CNN captures spatial information, RNN captures time or the ordering information (Words are one after the other). Also, graphs are a particular data structure and we can exploit their peculiarities to perform better predictions. Graphs are much more general: images can be seen as grid graphs (Fig. 4.5a), the same goes for a sequence (Fig 4.5b). Therefore GNNs are a generalization of both CNNs and RNNs. We also talked about "shallow" methods in the previous section. However, they also present some limitations, namely:

- We optimize directly the embeddings and not a set of parameters. By optimizing parameters we obtain a sort of "weight sharing". Two different node embeddings can be generated by the same set of parameters.
- They do not allow node features
- They are transductive, i.e., they can generate node embeddings only for nodes seen during training.

.



(a) Grid graph (images) **(b)** Sequence graph (text) **(c)** A random graph

Figure 4.5: Graphs are more general than images or text.

4.3.1 Graph Convolution

The main idea is to generalize the concept of convolution of CNN [14]. Since graphs have not a fixed notion of locality (e.g., An image has a top right corner), we cannot use a sliding window. The idea is to use message passing; each node will "send a message" to its neighbors. Each message is multiplied with weights taken from a weight matrix W. Finally, they are added up. We need to define how messages will flow; for this reason each node will have its **Computational Graph**. A computational graph is built starting from a target node and unfolding its neighbors for a predefined number of times. Each unfolding defines a layer of our GNN. (Fig. 4.6 shows a graph on the left side and the computational graph for node 1 on the right side.

It is also important to notice that, based on their neighborhood, each node will have its own computational graph. Each node will propagate messages to the next layer, and each time, they are first aggregated in some way and then processed by a neural network. The aggregation phase is of fundamental importance in GNN as it can be performed in different ways. As an example, we could simply average all the messages. We start with $h_0 = x_v$ where h_i represents the feature vector at layer i and x_v is the original input feature vector. The feature vector (or the embedding vector) at layer $i \neq 1$ is computed as:

$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \left(\frac{h_u^l}{|N(v)|}\right) + B_l h_v^l\right), \forall\, l \in \{0, ..., L-1\} \tag{4.4}$$

The σ symbol represents the activation function applied at the end. There are two

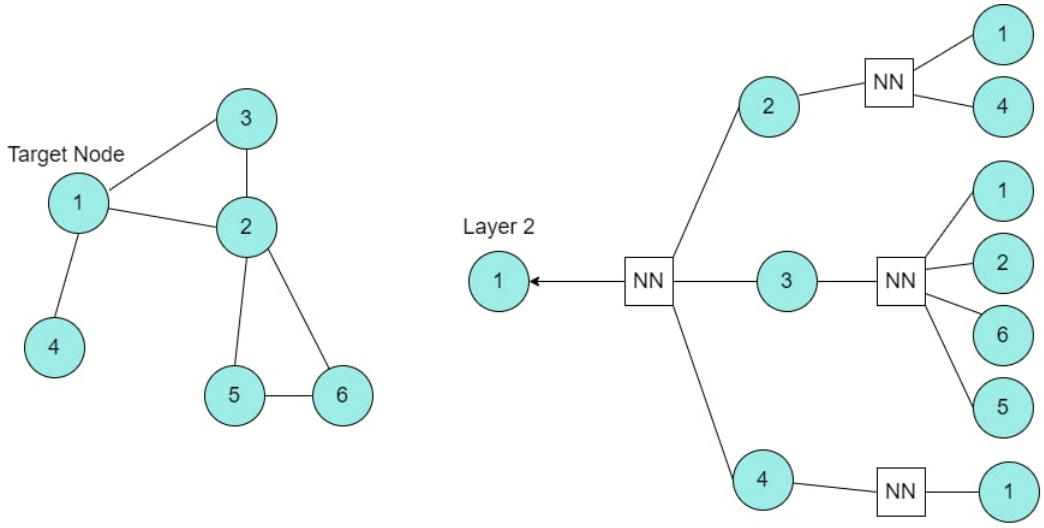


Figure 4.6: Computational Graph

matrices, W and B. W is the weight matrix for neighborhood aggregation. B is another weight matrix used to transform the target node embedding vector. Both matrices are trainable using any loss function and the gradient descent algorithm. It is important to notice that these matrices are shared between all nodes. The neighbor aggregation can be performed more efficiently if written in terms of matrix multiplication.

$$H^{(l+1)} = \sigma\left(D^{-1}A\,H^{(l)}\,W_l^T + H^l B_l^T\right) \tag{4.5}$$

- H^l is the embedding matrix, i.e., a matrix that contains all the node embeddings at a given layer l.

- A is simply the adjacency matrix.

- D is a diagonal matrix that contains the degree of each node. We use the inverse matrix of D so we have $1/degree$ in the diagonal.

- W and B are still the weight and bias trainable matrix.

This way, we obtain a product between matrices instead of a summatory.

4.3.2 Design choices in a GNN

When designing a Graph Neural Network, we are presented with a different set of choices. A GNN layer is composed of two operations: Message transformation and aggregation. Each different GNN architecture differs in how they define these two operations. Messages can also be transformed before being aggregated, for example, by multiplying them with a matrix. The aggregation operator must be permutation invariant. Also the way GNN layers are stacked is a design option. GNN layers can be simply chained together or skip connections could be added. When layers are chained, each layer l receives inputs from the corresponding layer $l-1$. A skip connections allows information to be received from any previous layer. Finally, the computational graph defined in the previous section can be augmented in order to learn better embeddings. Following, we will see more specifically the most famous types of GNN layers and design options.

Graph Convolutional Networks(GCN) [14]: Were one of the first proposed GNN layers, it is defined as:

$$h_v^l = \sigma\left(\sum_{u \in N(v)} W^l \frac{h_u^{l-1}}{|N(v)|}\right) \tag{4.6}$$

In this case the message is computed with $m_u^l = W^l \frac{h_u^{l-1}}{|N(v)|}$, that is at each layer l the message m_u^l computed by node u at layer l is the result of multiplying the embedding vector of the previous layer h^{l-1} by a weight matrix W^l. The result is then normalized by the size of that node neighborhood. The aggregation operation is the summatory; the messages from all neighbors are summed.

GraphSAGE [15]: . The GNN layer is defined as:

$$h_v^l = \sigma\left(W^l \cdot CONCAT\left(h_v^{l-1},\ AGG\left(\left\{h_u^{l-1},\ \forall\, u\ \in N(v)\right\}\right)\right)\right) \tag{4.7}$$

In this case the aggregation function has not a single definition. GraphSAGE allows different types of aggregation: Mean (Taking a weighted average of all neighbors), pool (transforming the neighbor vector and then applying the mean or max function) or more advanced techniques that we will not cover (Such as Long

Short Term Memory). Moreover, the message aggregation is done in two phases. First considering only neighbors of node v, that is what happens inside the AGG function. Then the result is concatenated with the embedding of node v itself.

Graph Attention Networks [16]: We would also like to cite this powerful kind of GNN. In both GCN and GraphSAGE all the neighbors of the target node are equally important. The idea of GAT is to add a learnable attention matrix $a_v u$ which allows the network to understand which neighbors are more important. GNNs allows many of the classical learning modules to be inserted between its layers. We will cover the **Batch Normalization** and the **Dropout** modules.

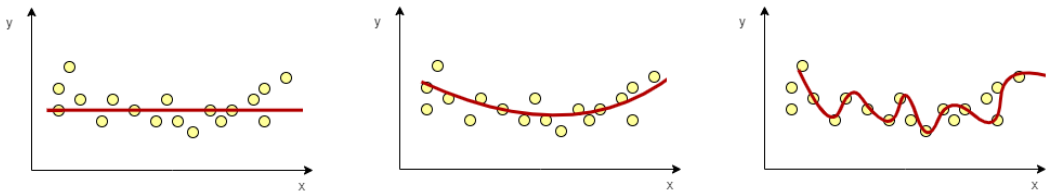
Batch Normalization [17]: When we are training a machine learning model, we map some input data to an appropriate output. Usually, is assumed that the input distribution remains the same for all the duration of the training. Since in Neural Networks each layer depends on the outputs coming from the preceding layer, this assumption may not hold. Batch Normalization helps stabilizing the networks by re-centering the node embeddings into zero mean and rescaling the variance into unit variance. The batch normalization is done in two steps: first we compute mean and variance over the embeddings then we normalize using the computed mean and variance. However, this normalization can reduce the expressive power of the neural network. To solve this problem, two learnable parameter are added into the formula so that the network is allowed to choose an arbitrary value for the mean and variance.

Dropout and overfitting [18]: What every machine learning model does is to train and tune its parameters using available data (the training dataset) but then apply the same rules on unseen data, i.e., they are useful only if they are able to generalize. When a model becames increasingly good in understanding the training dataset it might happen that the generalization power decreases. This is because often the training dataset has a different distribution than the original data. The more data we have the more we are able to get a good approximation of the real distribution. However, most practical cases have quite less data than needed. If trained too much, the model will learn to fit exactly the training data. This usually

causes bigger mistakes when evaluating the model with unseen data. A common example of overfitting can be seen when trying to fit a polynomial Dropout is a technique used to reduce the effect of overfitting in neural networks. The main concept is to remove a certain percentage of neurons when computing the output of a neural network layer. This avoids that during training, certain neurons become more important than others. The technique is divided in three phases:

- Assigning a dropout rate. This will control the percentage of neurons that will be removed.
- Remove random neurons until we reach the desired amount of neurons.
- The output is computed using the remaining neurons only.

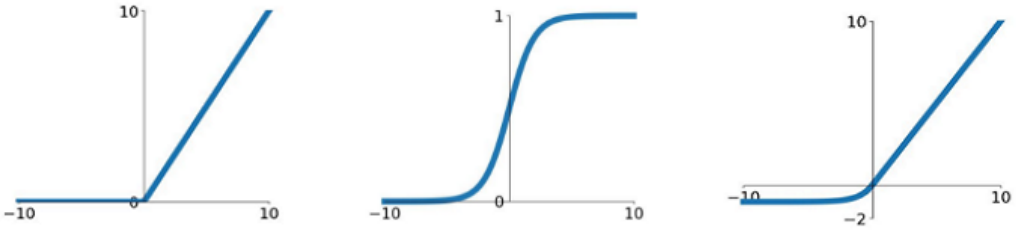
This way, all neurons are given a chance to contribute to the final output. In GNNs the dropout is applied in the message transformation phase.



(a) Underfitting **(b)** Balanced **(c)** Overfitting

Activation Functions: The most used non-linearities for GNNS are:

- Rectified linear unit (ReLU): defined as $ReLU(x_i) = max(x_i, 0)$. It is the most commonly used.
- Sigmoid: $\sigma(x_i) = \frac{1}{1+e^{-x_i}}$. Usually used when the range of our embeddings needs to be restricted.
- Parametric ReLU: adds a trainable parameter to the Relu: $PReLU(x_i) = max(x_i, 0) + a_i min(x_i, 0)$. Usually, it performs better than classical ReLu.



(a) ReLU **(b)** Sigmoid **(c)** Parametric ReLU

Graph Augmentation: As we said in the previous section, each node defines a computational graph. A computational graph is a graph that represent how the message passing happens for that target node. We explained how these computational graph can be built by simply unrolling a node neighborhood. However, this is not the only way to do it. It is unlikely that the vanilla computational graph is the optimal choice for computing embeddings. There are 4 problems with vanilla computational graphs. If the graph is too sparse, the message passing will be inefficient. This is solved by adding virtual nodes or edges to allow more message passing in the graph. A possible way to do it is to connect 2-hop neighbors using virtual edges. This is especially useful when working with bipartite graphs. It is also a good idea to add a virtual node that connects to all other nodes in the graph, such that all nodes will have a distance of two. This makes the graph much less sparse and it is beneficial for message passing. If the graph is too dense it might be beneficial to not consider the whole node neighborhood. Usually, at each layer we sample a different portion of a node neighborhood. This is done to reduce the computational cost of message passing when each node has a big neighborhood. Also, if a graph is too large there might be problems in fitting it on GPUs. Node features are also important when computing messages, as the first message at layer 0 is composed by the node feature vector. If nodes lacks features it might be useful to add them. A common approach is to simply assign a constant value like "1" to each node. This is especially useful in an inductive setting (when we want to generalize on unseen graphs). All nodes are identical but the GNN will learn from the graph structure. For transductive settings, a possibility is to assign to each node an ID and use it as a feature. This procedure is more expressive as node-specific information is stored but, since a GNN cannot know an unseen node

ID, it does not generalize well. If node features are not present, it is much harder for a GNN to differentiate between a nodes. As an example, the red nodes in Fig 4.9a and Fig 4.9b will generate the same computational graph (Fig 4.9c). To solve this problem, a possible augmented feature vector could be the cycle count. Each node knows the length of the cycle it resides in. This way, the red node in Fig 4.9a will have a value of 3 as the cycle it resides in is of length 3. The red node in Fig 4.9b will have a value of 4. Now the two nodes can be distinguished by a GNN since their feature are different (the structure of the computational graph will remain the same).

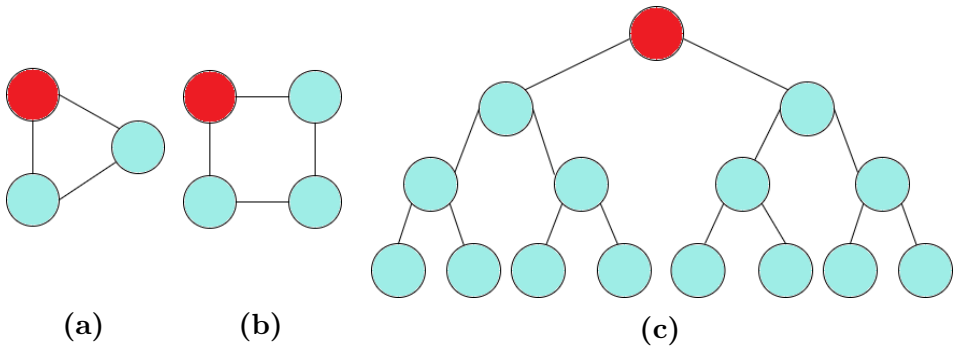


Figure 4.9: Red nodes in Fig 4.9a and Fig 4.9b will both generate the computational graph in Fig 4.9c.

Prediction setting: Using GNNs we can model different kinds of problems. It is useful to reduce a problem to known patterns or prediction tasks. The prediction task can usually be expressed in one of the following ways:

- Node-level: predict a discrete (classification) or continue (regression) value for nodes.
- Link-level: Classification or regression on node pairs or predicting new links.
- Graph-level: Classification or regression on the whole graph.

A GNN will produce embeddings at its last layer. For node-level tasks, we simply use the resulting embeddings to make predictions. For link-level tasks, we can generate new embeddings by concatenating the original embeddings in pairs. Alternatively,

we can simply use the dot product between two embeddings. If done this way we can simply predict a binary value that usually represents the existence or not of an edge. For graph-level tasks, embeddings must be aggregated in other ways. Common techniques consists in taking the mean , the max or the sum of all embeddings. Of course, by naively using these techniques, much expressive power is lost. Let's suppose we have G with a node embedding vector $z_g = [-1, -1, -2, 1, 3]$ and graph H with a node embedding vector $z_h = [-10, -10, -20, 10, 30]$ and we use the sum operation to aggregate these embeddings. Both G and H will have a resulting embedding of 30. Therefore, our GNN will not be able to differentiate between these two very different graphs. This can be solved by hierarchically aggregating the embeddings. For each graph we first aggregate the first two nodes, the result is fed to an activation function. The same is done with the last three nodes and the two final results are aggregated once again.

Setting up the dataset: Using a graph dataset is more tricky than the average dataset, e.g., an image dataset. In an image dataset, each image is an independent data point. The prediction on an image only depends on that specific image characteristics. When we want to classify nodes, the setting is different. Each node is connected with other nodes by one or more edges (Fig. 4.10). In this case, other nodes will affect the prediction on our target node. We can use our dataset in different ways depending on if our task is **transductive** or **inductive**. When we are dealing with a transductive setting we have only one input graph that is available in all the three phases (training, validation, test) of a neural network training pipeline. However, nodes in the graph are split in training, validation and test groups. When we are training, the embeddings is computed using the entire graph, but only the training nodes labels will be used. The same goes in validation, the evaluation is only done on the validation nodes. In the inductive setting the dataset does not consist on only one graph. The objective of the inductive setting is to generalize to unseen graph, not to predict missing node labels. In this case the dataset is split in training, validation and test just as we would do in an image dataset. There exists node-level or link-level tasks for both the inductive and transductive setting, however only the inductive setting is well defined for graph-level tasks.

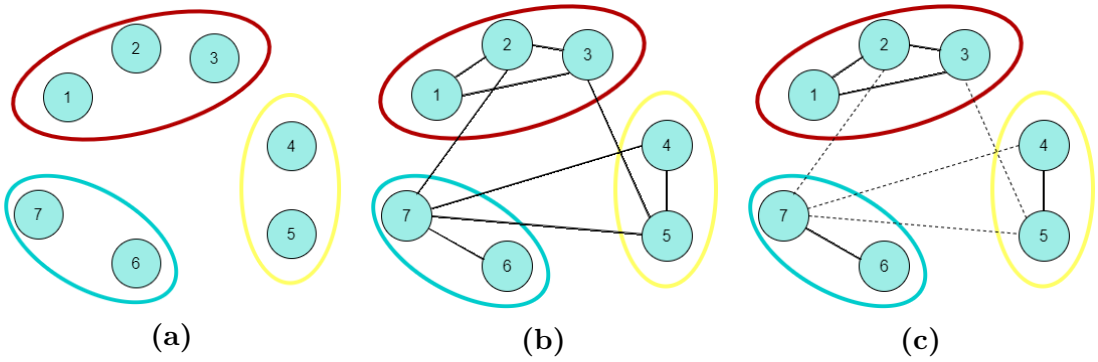


Figure 4.10: We divided 7 data points in training (red), validation (yellow) and test set (light blue). Figure (a) shows an image dataset where all points are independent. Figure (b) shows a graph dataset, where nodes are actually connected with each other. Figure (c) shows a graph dataset split in an inductive setting. The original graph is split into three independent graphs.

4.3.3 Expressive power of GNNs

The expressive power of a GNN is its ability to distinguish different graph structures [19]. In general a GNN is not able to distinguish nodes that are symmetric or isomorphic in the graph. For each node, its embedding is computed following the message passing paths defined by its computational graph. If two nodes have the same computational graph structure, also their embedding will be the same. Of course this is even more problematic if nodes have the same features.

The aggregation steps adds another problematic that could reduce the expressive power of the GNN. Two different set of messages can produce the same output after the aggregation. This results in different graph structures being confused for the same one.

Mean pooling: The mean operator used in Graph Convolutional Networks will produce the same output if the different labels have the same proportion among the inputs. As an example, let's suppose we have two labels, green and red, encoded respectively by the vectors (0,1) and (1,0). If before the aggregation step we receive two green messages and two red messages, by applying the mean for each element we obtain the resulting output (0.5, 0.5). The same output is obtained in any case where half messages are green and half messages are red.

Max pooling: The max operator, used in GraphSAGE cannot distinguish different structures where the messages are composed by the same set of distinct colors. Using the previous example, we have two colors, green and red, encoded by (0,1) and (1,0). If a message from both color is present, then the element-wise max pooling will always produce (1,1) as an output. Similarly, if only one color is present, it does not matter its quantity. If only red messages are processed by the max aggregation operator, the result will always be (1,0).

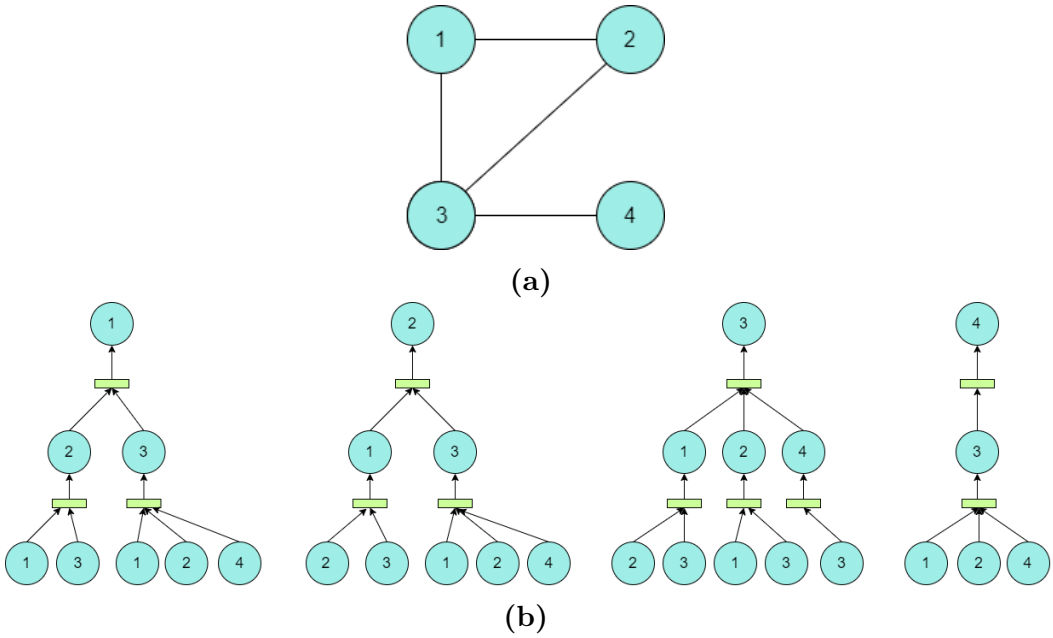


Figure 4.11: The two symmetric nodes 1 and 2 generates the same computational graph.

Chapter 5

GNN for the MCS problem

In this section, we propose three different methods to improve McSplit by using Graph Neural Networks. We recall that McSplit is an exact algorithm optimized to find the MCS between two inputs graphs in the shortest time possible. Since the MCS problem is NP-hard, this will hardly scale for bigger graphs. Our idea is different, we want to optimize the algorithm so that it can find a good solution in short budget time. To do this we propose three different methods that all use McSplit as a backbone. As previously mentioned McSplit makes use of two heuristics.

- Bidomain heuristic: Which select the bidomain with the smallest $max(|G|, |H|)$
- Node Ordering heuristic: Which prioritizes nodes with higher degree

The main objective of these heuristics is to reduce the tree search size, allowing more pruning to be done by the BnB algorithm. However, there are some drawbacks. The node degree is a too simple metric, many nodes have the same degree and therefore, most of the heuristic is based on random factors, i.e., the alphabetical order of nodes. Also, McSplit follows a "fail-first" paradigm. The "fail-first" paradigm is often used in BnB algorithms. The objective of such paradigm is to reach leaf nodes in the search tree as fast as possible so that a more aggressive pruning can be done afterwards. This is good for McSplit objectives since it is interested to the exact solution. We will start by introducing NeuroMatch (Section

5.1). Then using the representational power of Graph Neural Networks, we will first twist the node ordering heuristic with two "smarter" ones (Section 5.2). Afterwards, we will modify McSplit more drastically, introducing a best-first approach that selects node pairs directly (Section 5.3). Finally, we will present the idea of a custom neural network that learns a node's likelihood to be part of a big MCS solution (Section 5.4).

5.1 NeuroMatch

NeuroMatch is a graph neural network (GNN) which computes and enforces an order constraint on graph embeddings allowing a fast computation of graph matchings [20]. Given a graph $G = (V, E)$, NeuroMatch learns embeddings by selecting one at a time each node $v \in V$ and extracting its k-hop neighborhood using the Breadth-First Search (BFS) algorithm. Therefore, given v (also referred to as anchor-node) its embedding represents its k-hop neighborhood. We recall that we define the k-hop neighborhood of a given node $v \in V$ as the subgraph induced by the set of nodes that includes v and all nodes that can be reached from v with a path shorter than k. Notice that graph embeddings are learned enforcing an order constraint, as the relationship between subgraphs is represented by the geometry of the embeddings. This also allows computing matchings by just comparing the components of two embeddings.

NeuroMatch satisfies four properties necessary for subgraph relationships:

- *Transitivity*: if G is a subgraph of H and H is a subgraph of L, then G is a subgraph of L.

- *Anti-symmetry*: G is a subgraph H and H is a subgraph of G iff they are isomorphic.

- *Intersection set*: the intersection of the sets of G and H subgraphs contains all common subgraphs of G and H.

- *Non-trivial intersection*: the trivial graph, i.e., a graph with one node and no edge, is contained in the intersection between any two graphs.

In practice, a graph G which has an embedding vector Z_G with D dimensions is classified as being a subgraph of graph H with embedding vector Z_H if each component of Z_G is less than the corresponding component in Z_H.

$$\forall i \in [1, D] : Z_G[i] \leq Z_H[i] \iff G \subseteq H \tag{5.1}$$

To ensure the already mentioned order constraint, NeuroMatch is trained using the max-margin loss:

$$\begin{aligned} \mathcal{L}(Z_G, Z_H) = \textstyle\sum_{(Z_G, Z_H) \in P} E(Z_G, Z_H) + \\ \textstyle\sum_{(Z_G, Z_H) \in N} \max(0, \alpha - E(Z_G, Z_H)) \end{aligned} \tag{5.2}$$

where $E(Z_1, Z_2) = \max(0, |Z_1 - Z_2|_2^2)$. P and N are respectively the positive and negative samples. Positive samples are graph pairs in which the first graph is a subgraph of the second. The pairs in which this condition does not hold are defined as negative samples. Loss is minimized when the subgraph relationship, defined by E, holds in the case of pairs in P and it is violated by at least α pairs in N. The training procedure is further enhanced by a curriculum learning scheme, i.e., the model is first trained on easier instances which progressively gets harder when the model loss stabilizes. As we can see in 5.1, the result is an order embedding space. A graph G is contained in a bigger graph H if G is found in the lower-left side of H in the order embedding space.

Embedding Computation

In our specific scenario, Embeddings for each node $v \in G$ are computed considering their k-hop neighborhood. Sampling all the neighbors up to k leads us to the associated induced subgraph of G, on which the embedding is computed. To speedup the process by exploiting the GPU parallelism capability, we divide the sampled subgraphs in batches before loading them into the GPU memory. This allows the GPU to spend less time in an idle state, since each thread warp is generated by multiples of 32 threads and having only one thread working means that 31 threads remain idle, nullifying the advantages of the GPU. NeuroMatch does not rely on a specific GNN and it can exploit all the different state-of-the-art models such as the Graph Convolutional Network (GCN), GraphSAGE, or Graph

Isomorphism Network (GIN). In our flow, we use an 8-layer GraphSAGE model.

Selecting a Proper Depth for the BFS

Given an anchor-node v, the value of the parameter k allow us to control the depth of the breadth-first visit we use to build our k-hop neighborhood subgraphs. In other words, k can be seen as the eccentricity of the anchor-node in the sampled subgraph. At the same time, we use a GNN with a predefined number j of layers. It is worth noticing that j and k have no relation for the following reason: A GNN with j layers computes an embedding for a specific node by aggregating and transforming information gathered from its j-hop neighbors. Since we first sample a k-hop neighborhood subgraph using a BFS, if we had $k > j$ this would imply that all nodes at a distance $d > j$ from the anchor vertex would not be considered. Therefore, the number of layers j has to be always larger or equal to the value of k to ensure that the information coming from all nodes will propagate to the anchor.

The value of k also affects the overall performance. Indeed, even if during the embedding computation the number of layers is j, the value of k coincides with the depth of the BFS procedure we need to run, and subsequently k increases the time cost of the procedure.

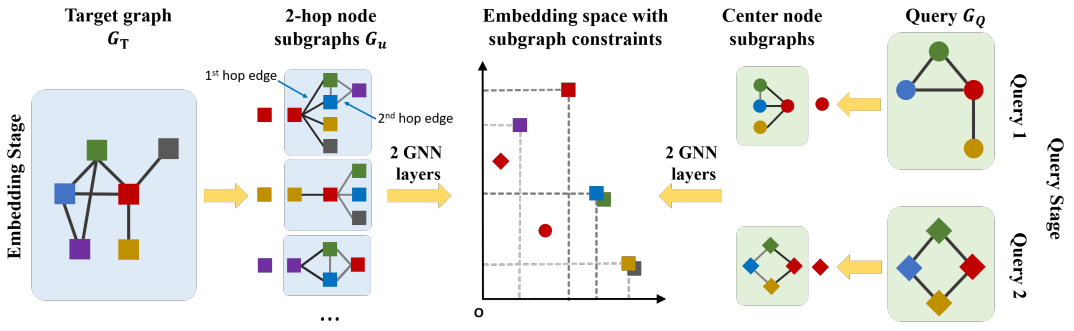


Figure 5.1: NeuroMatch architecture, image retrived from http://snap.stanford.edu/subgraph-matching/

5.2 GNN based heuristics

As introduced at the beginning of this chapter, McSplit first selects the bidomain with the smallest $max(|G|, |H|)$, i.e., by comparing the values delivered by the formula for all possible bidomains. Moreover, within the selected bidomain, it chooses the vertex with the highest degree. Essentially, given the graphs G and H, the algorithm creates the final $MCS(G, H)$ by matching one vertex of G with one vertex of H, sorting the nodes of G and H based on their degree, such that high-degree nodes are paired before along the branch-and-bound process. Albeit very simple, this strategies presents some drawbacks, mainly due to the fact that the node degree does not capture graph structural information on neighbors and it leads to many ties finally solved using the original lexicographic (and meaningless) order of the nodes. This leads to scalability issues and inefficiency on large graphs. Our idea is to exploit the McSplit core ideas and change its static heuristic by ranking nodes using *NeuroMatch embedding norms* and *Cumulative Cosine Similarity*. Given a graph $G = (V_G, E_G)$, for each node $v \in V_G$ (the "anchor" node), we consider a subgraph including all nodes in the k-hop neighborhoods of v. Increasing or decreasing the value of k, we agglomerate into the node embedding more fine-grained or coarse information on G. Norms compactly represent information regarding the order and the size of each subgraph obtained by k-hop partitioning the graph. Larger subgraphs are associated to higher embedding norm values. The Cumulative Cosine Similarity is obtained by summing the cosine similarity, gathered by comparing NeuroMatch embeddings, between a node in the first graph and all other nodes in the second graph. This has the effect to rank a node higher when it has many corresponding nodes in the second graph, having an higher a priori probability of being part of a bigger solution.

With Algorithm 1 as a reference for McSplit, our heuristics replaces the node degree heuristic in both lines 9 and 11. Therefore, the priority scheme considers the maximum norm or the cumulative cosine similarity instead of the maximum degree. Notice that as in the original algorithm, our scores are computed during an initialization step before calling the mcs function represented in the pseudo-code.

Algorithm 1 mcs$(G, H, M, incumbent, L)$

Input : Two graphs $G = (V_g, E_g)$ and $H = (V_h, E_h)$; $incumbent$, the biggest common subgraph found thus far; M, the solution being built; L the current set of labels (bidomains)

Output : $incumbent$

```
1  if |M| > |incumbent| then
2  |  incumbent ← M
3  end
4  bound ← |M| + calc_bound(L);                                  // Eq. 3.1
5  if bound ≤ |incumbent| then
6  |  return incumbent
7  end
8  bd ← select_bidomain(L);                                      // bidomain selection
9  v ← select_left_node(bd);                                     // node selection
10 remove v from bd
   for each w ∈ bd do
11 |  remove w from bd
   |   M.push((v, w));                                          // Add new pair to M
12 |  new_L ← filter_labels(L, v, w);                           // Generate new L set
13 |  incumbent ← mcs(G, H, M, incumbent, new_L)
   |   M.pop()
   |   add w to bd;                                             // w is added back
14 end
15 if bd.V_{l,g} is empty then
16 |  remove bd from L
17 end
18 return mcs(G, H, M, incumbent, L)
```

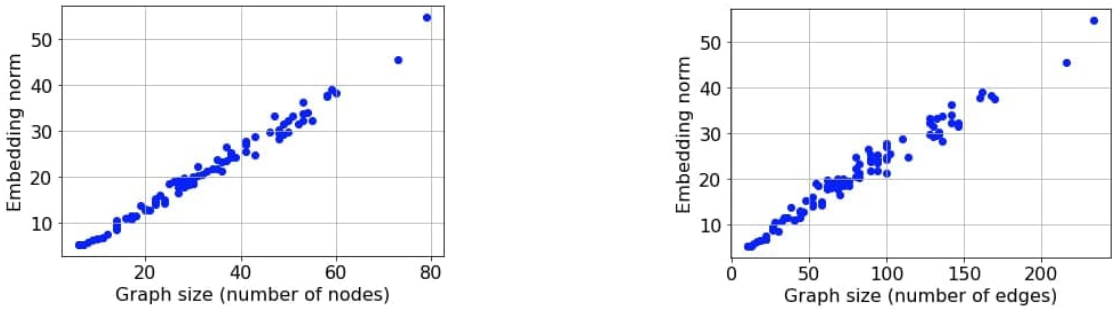
5.2.1 Embedding norm

Our first heuristic is based on the L2 norm. We compute the Euclidean Norm from NeuroMatch embeddings evaluated on the k-hop neighborhood of each node in both graphs. More specifically, we compute:

$$L2\left(Z\right) \quad = \quad \sqrt{\sum_{i=1}^{D} Z\left[i\right]^2} \tag{5.3}$$

where Z is a D-dimensional embedding vector. The L2 norm of such embeddings is higher for nodes of big and dense graphs, due to the NeuroMatch order constraint.

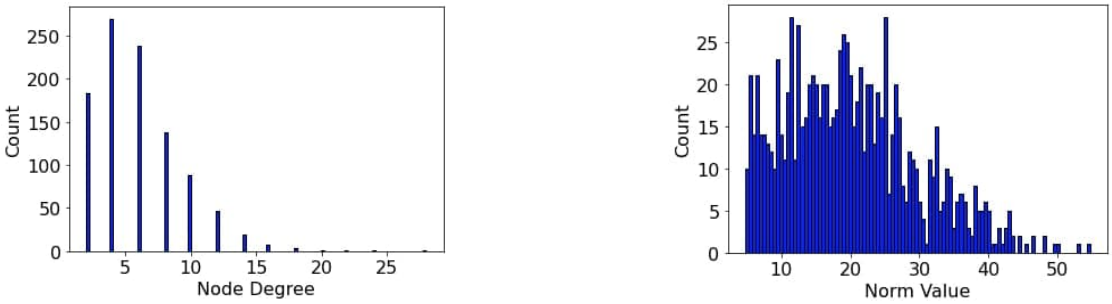
Therefore, instead of simply considering a node degree we take into consideration both the size and the order of the whole k-hop neighborhood. The L2 norm of such embeddings is higher for nodes of big and dense graphs, due to the NeuroMatch order constraint. Therefore, instead of simply considering a node degree we take into consideration both the size and the order of the whole k-hop neighborhood. To prove this conjecture, we run some experiments and we report our analysis in Figure 5.2. We sample 100 subgraphs from a single graph with 100 nodes by using a BFS with a depth limit equal to 3. Then, we plot the norm value against the number of nodes (Figure 5.2a) and the number of edges (Figure 5.2b) for each graph.



(a) **(b)**

Figure 5.2: The embedding norm grows higher with both the number of nodes (Fig. a) and the number of edges (Fig. b)

Figure 5.3a and 5.3b show the distribution of the degree and of norms respectively computed on a set of 10 graphs with 100 nodes each. It is possible to observe that the distribution of the node degree is much less sparse. Since the graph is connected and it has 100 nodes, the node degree values could range from 1 to 100. On the contrary, most nodes tend to have few neighbors and the mean degree is about 5. This is a common situation in the graph domain as nodes tend to be aggregated in little neighborhoods. For the embedding norm we could have a different value for each possible different subgraph composed by 100 or less nodes. Moreover, the degree of a node is a discrete variable while the embedding norm is continuous. Then, since McSplit breaks ties lexicographically, the performance of the node degree heuristic is much more dependent on random factors.



(a) **(b)**

Figure 5.3: The node degree distribution (Fig. a) is much less sparse than norm value distribution (Fig. b)

Recomputing the Norm

Since we try to give higher priority to nodes with a bigger and denser neighborhood, it might be useful to recompute the norms from time to time while the procedure is running. In this way, it is possible to remove already selected nodes from all neighborhoods. Unfortunately, when we recompute the norm at each recursion step, the computational cost overhead is so high that recomputing norm covers the majority of our budget time. Thus, even if this strategy should maximize the efficiency of the selection heuristic, and reach good solutions with less recursions, we need to improve it from the computational point of view to make it appealing. McSplit selects a new pair of nodes at each recursion call. Thus, at any new call along the search tree, the new pair of nodes is added to the previously selected set of pairs. Previously selected or non-selected pairs are not considered anymore when making a new selection and growing the current solution. This consideration has some influence on how and when we actually compute (and possibly update) our embeddings. By computing embeddings only at the beginning, we are considering the full k-hop neighborhood around each node. However, as we go on with the algorithm, it could be smarter not to consider nodes already in the solution as part of any neighborhood. For example, it is possible to have the following situation. A vertex v has a large neighborhood but this contains mainly nodes already included in the solution, whereas a vertex u has a small neighborhood but this incorporate mainly nodes still available for future matchings. In this condition, the norm evaluated at the beginning of the process can suggest a wrong vertex selection by

prioritizing v with respect to u.

To avoid this problem, the most obvious solution is to recompute embeddings (potentially) after each pair selection and, at the same time, ignoring all nodes already selected in the current solution when sampling neighbors in the BFS procedure. This solution corresponds to removing all nodes in the current solution from the previously sampled subgraphs. Therefore, the nodes affected by this procedure will have smaller norms and a smaller priority moving along the selection process. At the same time, this process will increase the cost to compute embeddings.

Experimentally speaking, computing the embeddings only at the beginning of the process cost only about 0.1-0.4% of the total budget time t. On the contrary, recomputing the norms at each recursion can take up over 95% of t, dramatically reducing the total number of recursions the algorithm can perform. To trade off time and accuracy in norm computation, a compromise is therefore in order. One possible strategy, is to recompute the norm if the algorithm does not improve the solution size after a pre-defined time, or a predefined number of recursions, is reached. This could be done in different ways. The naive approach would be to simply stop the algorithm at any branching point, recompute the norm, and restart the algorithm with the new norms. A better approach could be to first backtrack to a promising branching point and then computing norms there. For example, in [21], the algorithm backtracks to the branching point with the largest action space, i.e., the one with the highest number of possible branches. This can also help to avoid falling into local optimum.

5.2.2 Cumulative Cosine Similarity

As in the previous section, our objective is to compute a score for each node in both graphs. The cosine similarity between two nodes represent the similarity of their respective k-hop neighborhood. Nodes with very similar neighborhoods will have an higher similarity score. Thus, in this case, we sort the nodes of the graph based on the sum of the value of the cosine similarity computed between one node of G and all nodes of H. The main advantage of considering this metric, is that the score of each node is obtained by reasoning on both graphs, whereas the heuristics based on the norm and the degree consider only a single graph.

In Figure 5.4 and 5.5 we show the cosine similarity values for two pairs of graphs.

The more two graphs are similar, the higher the cosine similarity. Given two graphs G and H, in the first step we compute a matrix M where M_{ij} is the cosine similarity score between node $i \in G$ and node $j \in H$. Then, the score of each node in G is given by the sum of values on the corresponding row in M, while the scores of nodes in H is given by summing over columns. Specifically, for the nodes in G we compute:

$$CSS\,(i) = \sum_{j=1}^{|H|} M_{ij} \; where \; M_{ij} = Z_i \cdot Z_j \tag{5.4}$$

Z_i and Z_j are respectively the embedding vectors of node $i \in G$ and node $j \in H$. This score reflects how frequently a subgraph, sampled from one graph and represented by a node k-hop neighborhood, is found inside the other graph. These nodes have an higher probability to be part of a big solution or to be matched in general, therefore we give these node an higher priority.

(a) **(b)**

Figure 5.4: Two graphs with a cosine similarity of 1, note that they are isomorphic

(a) **(b)**

Figure 5.5: Two graphs with a cosine similarity of 0.22, note that they are quite different

Neighborhood sampling and cosine similarity

The number of GNN layers have a large impact on how informative the cosine similarity is. It can be shown that a GNN expressive power does not monotonically increase with the number of layers, as it happens in other kind of neural networks. This is a phenomenon called over-smoothing
[22], and it also depends on the graph size. Usually, bigger graphs allows more layers to be added before reaching over-smoothing. In our particular case, we keep the number of layers fixed, but we sample a node k-hop neighborhood from the original graph before embedding it. (Fig 5.6) shows that this problem also arises in our setting. The dataset used in the figure contains graphs up to 100 nodes. Our analysis leaded us to the conclusion that a value of k of 1 or 2 could be optimal for this dataset.

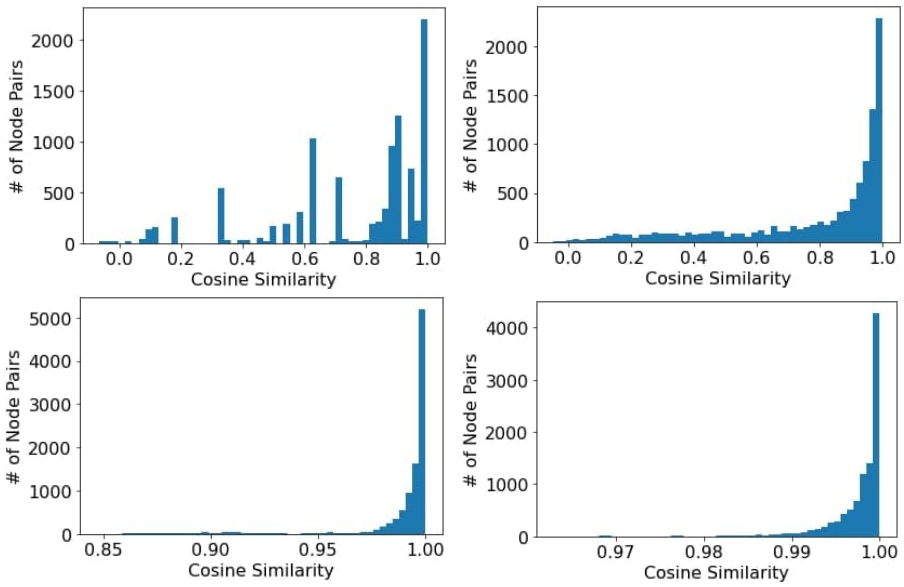


Figure 5.6: Distribution of cosine similarity scores computed for all node pairs in a 100 nodes sample graph considering its k-hop neighborhood with k equal respectively to 1, 2, 4 and 6 (Clockwise from top left). Note that for k equal to 4 almost all pairs have a similarity greater than 0.9, and for k equal to 6 almost all pairs have a similarity greater than 0.99

5.3 Best-pair McSplit Implementation

In this section, we propose a more drastic modification of the McSplit original algorithm, which instead of using two heuristics, uses only one. The used heuristic is again based on the cosine similarity. While McSplit heuristic works at node-level, we substitute the node selection with a node-pair heuristic, that is, we give scores directly to node pairs. In McSplit, a node from the first graph is selected, and then paired with each node of the second graph. The nodes of both graphs are prioritized following the same heuristic. McSplit also subdivides nodes in different classes, which represent a node connectivity pattern with respect to already selected node pairs. These classes are used to further prune the search space. Before selecting nodes using the node-degree heuristic, McSplit uses another heuristic to select the class from which nodes are considered. Instead of sorting nodes, we give a different priority directly to node pairs. The score we give to each pair, is the cosine similarity computed from the embeddings of both nodes. We first try to select a pair belonging to a node class, chosen by McSplit class heuristic. Subsequently, we further modify the algorithm in order to select the best pair among all classes, therefore, substituting both McSplit original heuristics with our node-pair heuristic. We still use McSplit classes to prune the search space.

Selecting Most Similar Pairs in a Selected Bidomain

Without touching the McSplit bidomain heuristic, we prioritize node pairs with higher cosine similarity, therefore, replacing only the node ordering heuristic. To make it possible, it is necessary to modify some parts of McSplit. McSplit original implementation selects one node from the first input graph, and tries to match it with all nodes from the second input graph that are in the bidomain chosen by the bidomain heuristic. In our case, we directly selects node pairs. Once the bidomain is chosen, we try all node pairs in that bidomain, prioritizing pairs with high cosine similarity scores. Cosine similarity scores can be pre-computed, and stored in a priority queue. However, since we don't know which bidomain will be chosen a priori, we need to select a subset of the queue, corresponding to all node pairs in the chosen bidomain, at each recursion level. Additionally, as we did with the norms heuristic, we try to recompute the node embeddings at each recursion call.

The new embeddings are still computed considering the nodes k-hop neighborhoods but excluding already selected nodes.

Selecting Most Similar Pairs Among All Bidomains

In this case, we no longer consider the McSplit bidomain heuristic, but we will directly select a node pair. The chosen bidomain is automatically derived by the fact that each node pair belongs to a single bidomain. Before the recursive phase begin, we compute the cosine similarity among all node pairs and store them in a priority queue. Again, node pairs with higher cosine similarity are prioritized. The same copy of the priority queue is used in all recursion level. To allow this, we keep track of the last chosen pair index in the queue. That is, if the last picked pair was at index i, the first pair which is considered in the next recursive call is the one at index $i + 1$. This also avoid to consider permutations of the same solution. Note that, while the algorithm goes on, not all pairs remains valid. We can only pick pairs which are in the same bidomain in that particular recursion level, so, even if we consider first the pair at index $i + 1$ does not mean that pair will be picked. In such case, the next pair in the queue is considered until a valid pair is found. Algorithm 2 shows the full procedure.

The get_pair function finds a valid pair in the queue starting from index idx. To check for a valid pair is necessary to know to what bidomain each node belong. This information can potentially change at each recursion level. To make checks more efficient, a dictionary which maps each node to its bidomain its computed.

5.4 Custom Neural Network

All previous attempts focuses on learning a score, on node or node-pairs, which is able to correctly prioritize nodes, so that, we reach a bigger solution faster. However, they are all based on an **unsupervised** machine learning approach. We first learn node embedding from the graphs structure, and then we use them by computing norms or cosine similarities to obtain such scores. In this section, we will propose a **supervised** learning approach to learn these scores. Unsupervised learning is based on learning embeddings directly from unlabeled the data, i.e., it learns from the data structure only. Supervised learning learns from labeled data.

Algorithm 2 mcs($G, H, M, incumbent, L, queue, idx$)

Input : Two graphs $G = (V_g, E_g)$ and $H = (V_h, E_h)$; $incumbent$, the biggest common subgraph found thus far; M, the solution being built; L the current set of labels (bidomains); queue, priority queue with cosine similarity for each pair; idx, first index to consider at this recursion level

Output : $incumbent$

19 **if** $|M| > |incumbent|$ **then**
20 $incumbent \leftarrow M$
21 **end**
22 $bound \leftarrow |M| + calc\_bound(L)$ **if** $bound \leq incumbent$ **then**
23 **return** $incumbent$
24 **end**
25 **while** $idx < queue.size()$ **do**
26 $new\_idx, v, w, bd \leftarrow get\_pair(queue, idx, L)$; // Get next valid pair, its index and its bidomain
27 remove v and w from bd
 $M.push((v, w))$; // Add new pair to M
28 $new\_L \leftarrow filter\_labels(L, v, w)$; // Generate new L set
29 $incumbent \leftarrow mcs(G, H, M, incumbent, new\_L, queue, new\_idx)$
 M.pop()
 add w and v to bd; // w and v are added back
30 $idx \leftarrow idx + 1$
31 **end**
32 **return** incumbent

Our objective is to learn score for node-pairs, these scores are thought to be used together with the McSplit best-pair variation we proposed in Section 5.3. The idea is to use a training dataset composed of a set of MCS problem instances, together with the optimal solution (computed by the original McSplit). Our network should then learn to compute embeddings, such that, our notion of similarity between two nodes (being paired in a McSPlit solution) can be represented as a similarity between embeddings vector, i.e., two nodes that have been paired in optimal solution, should be close together in the embedding space. If two nodes have not been paired, they should be far from each other in the embedding space. The similarity measure will be a value between 1 and 0, and it will be interpreted as the probability of a pair of nodes to be selected by McSplit as part of the optimal solution. If correctly learned, by prioritizing these pairs, our best pair version of

McSplit should converge faster to the optimal solution. The network is composed by two MLPs (Figure 5.7). We use both graphs as an input, and they are both fed to the first MLP separately. Subsequently, the MLP outputs are joined and fed to the second MLP. The final output is a likelihood matrix. The major limitation of this approach, is the use of the MLP model, which requires a fixed size input and is not optimized for graphs. In future works, this limitation could be surpassed by using Graph Neural Networks.

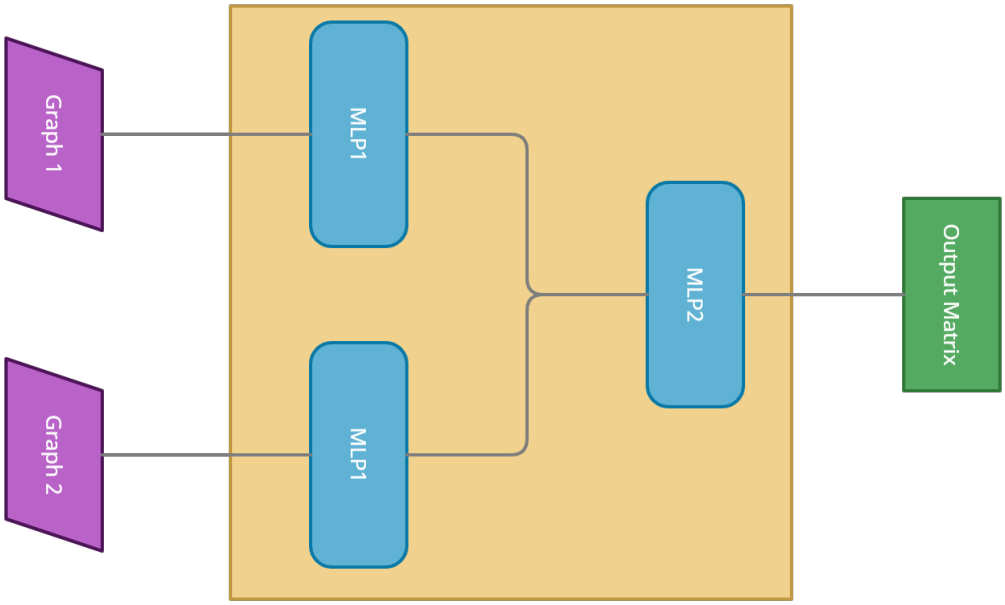


Figure 5.7: Our custom neural network architecture.

A graphical representation of our architecture can be seen in Fig 5.7. Algorithm 3, shows the python implementation. We used PyTorch as a deep learning framework. The dim value represents the size of the input graph. For example, if we decide to analyze graphs with 10 nodes, dim will be equal to 10, while the adjacency matrix will be of dimension 10*10. The output likelihood matrix, could be used in combination with our Best Pair McSplit implementation described in the previous section. In fact, Best Pair McSplit could prioritize nodes with a higher likelihood of being paired before others, potentially getting really close to the optimal solution in its first attempts.

Algorithm 3 Python implementation of our custom neural network

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = torch.nn.Flatten()
        self.linear_relu_stack = torch.nn.Sequential(
            torch.nn.Linear(dim*dim, 1000),
            torch.nn.ReLU(),
            torch.nn.Linear(1000, 1000),
            torch.nn.ReLU(),
            torch.nn.Linear(1000, dim*1000),
        )
        self.common_part = torch.nn.Sequential(
            torch.nn.Linear(dim*1000*2, 1000),
            torch.nn.ReLU(),
            torch.nn.Linear(1000, 1000),
            torch.nn.ReLU(),
            torch.nn.Linear(1000, dim*dim)
        )
    def forward(self, X):
        x1 = X[0]
        x2 = X[1]
        x1 = self.flatten(x1)
        logits1 = self.linear_relu_stack(x1)
        x2 = self.flatten(x2)
        logits2 = self.linear_relu_stack(x2)
        cat = torch.cat((logits1, logits2), dim = 1)
        logits = self.common_part(cat)
        return logits
```

Chapter 6

Results

6.1 GNN based heuristics

We compare these implementations (each one with several settings) with the original sequential versions of McSplit[1] and McSplit+RL[2].

Our implementations are written in C++, the same language of the original ones and, instead of sorting the nodes using their degree, they load an arbitrary score vector which is then used to pair nodes, The score vectors are pre-computed adopting a Python script, which is able to exploit the Pytorch framework, the Pytorch Geometric, and the DeepSnap libraries used in NeuroMatch[3]. With this approach, we pair the efficient neural network environment embedded in the Python framework with a faster recursive language (such as the C/C++). This is important because the Python implementation of the original algorithm, which is the most time consuming part of the entire process, is from one to two order of magnitude slower than the original one. The instances were retrieved from the original GitHub repository of McSplit[4]. All our experiments run on a i9 9900K CPU coupled with an NVIDIA GTX 1060 GPU for embedding computation. We compile our code

[1] `https://www.github.com/jamestrimble/ijcai2017-partitioning-common-subgraph`

[2] `https://github.com/JHL-HUST/McSplit-RL`

[3] `https://github.com/snap-stanford/neural-subgraph-learning-GNN`

[4] Folder *mcs-instances* in `https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph/`

on an Ubuntu 20.04 operating system, running through a Window Subsystem for Linux version 2. For the sake of fairness, the experiments were conducted on two sets of graphs:

- 50 synthetic, connected, unlabeled, and undirected graph pairs of small size (less than 50 nodes) on which the MCS can be found by at least one of the strategy.
- 50 graph pairs with similar features but of medium size (from 50 to 100 nodes) on which none of the methods is able to find the MCS in the slotted amount of time.

We set a time limit of 50 minutes for all runs. Moreover, as suggested by [20], to effectively train NeuroMatch, we selected a few hyper-parameters. Following their analysis, our NeuroMatch model uses an 8-layer GraphSAGE and an embedding space with 64 dimensions. This looks like being the configuration that maximizes the accuracy, i.e., the number of correct matchings performed by NeuroMatch. We also run experiments with the parameter k ranging from 1 to 8. As we also fixed the number of GNN layers to 8, larger values of k would not change the results as nodes at distance higher than 8 would simply be ignored.

The norm heuristic

Table 6.1 presents the results of our norm heuristic considering all values of k for the set of small size graph pairs. For these benchmarks all algorithms are able to finish the elaboration in the slotted elapsed wall-clock time. The table reports the number of instances in which our version wins against McSplit and McSplit+RL. Since every method finds the optimal solution, we compare methods based on the number of recursions performed. Note that these instances are fairly easy to solve and, as showed in [9], McSplit+RL does not perform well on them.

Table 6.2 shows our results on instances where at least one of the methods run out of time. The table is presented using the same format and the same notation of Table 6.1.

In this case, to establish a winner, we consider the size of the solution first and then the number of recursions required to compute it. More specifically, if

	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
McSplit	26	25	22	21	23	23	23	23
McSplit+RL	30	26	27	23	25	25	25	25

Table 6.1: Comparison of McSplit original heuristic and McSplit+RL against our norm heuristic with different value of k on smaller graphs.

	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
McSplit	24	25	27	26	20	21	24	24
McSplit+RL	27	26	28	26	22	23	26	25

Table 6.2: Number of wins of our norm-based strategy over McSplit original heuristic and McSplit+RL on larger graphs.

a method finds a solution larger than the other, it is considered as the winner. Otherwise, when the two results have the same size, we break ties by considering the number of recursions made to reach that solution size for the first time. The advantage of our method decreases with all values of k for smaller graphs, while for larger graphs the number of wins oscillates around half the number of of the graphs. As we can see, even if a method is able to explore the whole search space faster this does not necessarily imply that it can find a better solution in less time. Results show that norm heuristic is able to improve over McSplit in around half of the cases for each value of k both on smaller and on larger graphs.

Table 6.3 reports the number of cumulative wins of our procedure against the original version of McSplit (which appears to be more efficient than McSplit+RL in our experiments) on the large graph pair set, given all eight different values of k. The cumulative wins in percentage shows the potentiality of the method. Indeed, recalling that there are several parallel implementation of McSplit [23], the cumulative result can easily be reached by running several instances of McSplit in parallel, each version adopting a different value of k. Another possibility for using less resources is predicting the optimal value of k for each instance. Intuitively, since k represents the hop "distance" considered from the selected node, there may be a correlation between k and the size of the graph. Unfortunately, we have not been able to find any relation to such measures so far.

Since k controls the depth at which the k-hop neighborhoods are sampled, high

	k=1	**k≤2**	**k≤3**	**k≤4**	**k≤5**	**k≤6**	**k≤7**	**k≤8**
Wins	24	38	42	43	43	44	44	44
[%]	48	76	84	86	86	88	88	88

Table 6.3: Cumulative (from left to right) wins of norm heuristic against McSplit original heuristic with different values of k.

value of k can perform better on larger or denser graphs. Therefore, a first idea is to look-for some correlation between k and some graph measures, such as the graph order, the average number of edges, the density, or the diameter. Unfortunately, we could not find any relation to such measures. In Table 6.4 we compare our heuristic with norm recomputation, i.e., we recompute the norm at each recursion, against McSplit.

Recomputing the norm at each step does not help either, since the algorithm results up to 30,000 times slower, without providing any significant improvement.

We tried to recompute norms at each recursion step. The test was performed with a value of k equal to 1. Unfortunately, adopting the current implementation, that is far from being optimized, recomputing the norm makes the algorithm slower by a ratio of 30000 but it didn't show improvement against either McSplit original heuristic and our norm heuristic. For the evaluation we considered a prefixed number of recursions, around 5K, and considered as a winner the method which finds the biggest common subgraph.

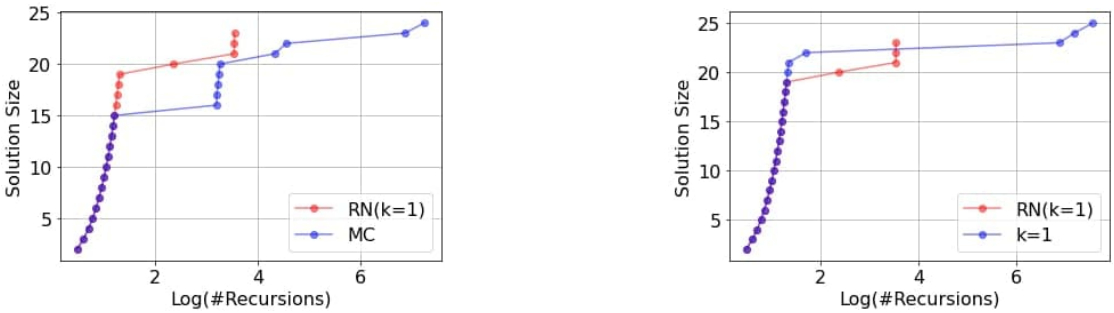
Generally the McSplit Python re-implementation is capable of reaching 80M recursions in the pre-fixed 10 minutes time budget. This is with either its original heuristic or ours. When trying to re-compute norms at each recursion step, we can only reach about 3K recursions, therefore the algorithm is about 30000x slower. For that reason the table reports the comparison considering an equal number of recursions, around 5K. Since even considering the number of recursions only the two approaches seems tied, we decided to not further investigate in this direction as the cost of the norm recomputation is too high.

Figure 6.1 compares our heuristic while recomputing the norm against McSplit original heuristic and our heuristic without recomputation on the same problem instance. The norm recomputation heuristic is at disadvantage against both, but this

Baseline	Wins	Losses
k=1	36	34
McSplit	30	40

Table 6.4: Comparison of both McSplit original heuristic and our norm heuristic without recomputing the norm (with k=1), and our heuristic when recomputing the norm at each recursion step (k still equal to 1).

is also due to its much slower speed. Considering less than 10000 recursions, which is more than three times the total number of recursions the norm recomputation heuristic can do in 10 minutes, it is at advantage.



(a) **(b)**

Figure 6.1: Comparison between our norm heuristic with and without norm recomputation. Our heuristic with norm recomputation (RN) outperforms McSplit original heuristic up to a certain number of recursions. After that number, norms may lose their power and the original heuristic may be faster to converge.

Cumulative Cosine Similarity

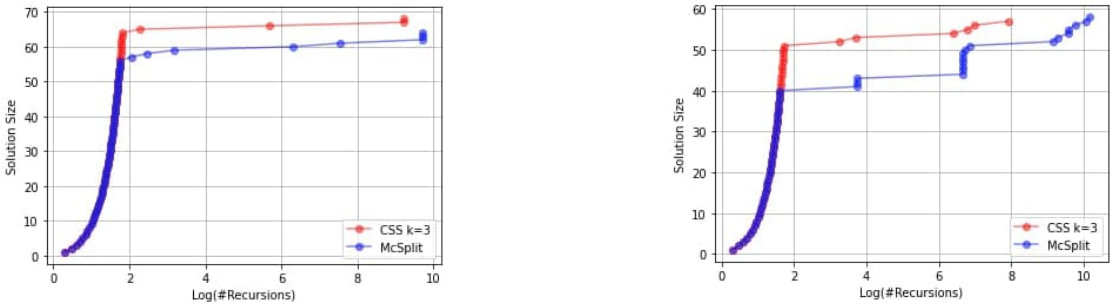
Table 6.5 compares our cumulative cosine similarity heuristic against McSplit and McSplit+RL for the smaller graph pair set. In this case, the norm heuristic outperforms the cumulative cosine similarity. Within this method, scores are computed considering information from both graphs instead of a single node only.

Figure 6.2 shows an in-depth analysis of two instances of the problem, of our cumulative cosine similarity heuristic against McSplit, reporting the solution size (on the y-axis) as a function of the number of recursions (x-axis). For the sake of

	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
McSplit	11	15	19	16	17	17	17	17
McSplit+RL	16	18	19	17	17	18	18	18

Table 6.5: Number of wins of our cumulative cosine similarity-based strategy over McSplit original heuristic and McSplit+RL on small graph pairs.

readability, the number of recursions (reaching a value of 74B in the first case and of 73B in the second one) is reported on a logarithmic scale. Each dot of the two graphics represents the first time the related heuristic reaches a specific solution size. We can see that the cumulative cosine similarity improves over the original algorithm by finding larger solutions in a shorter amount of time.



(a) **(b)**

Figure 6.2: Our cumulative cosine similarity (k=3) heuristic against McSplit.We can see how larger solutions are found in a shorter amount of time. In the first case (left) our heuristic (red) finds a solution of 68 nodes, whereas the original procedure (blue, MC) stops at 64 in the same amount of time. In the second one (right) our heuristic finds a solution of size 57 whereas McSplit gets to 58 nodes in the same slotted time of 50 minutes.

Table 6.6 shows the number of wins of our cumulative cosine similarity heuristic on the large graph pair set. We can clearly see the advantages of this heuristic, as it also outperforms the norm-based strategy. This is due to the behavior shown in Figure 6.2 and described earlier. In this case, however, differently from the norm heuristic, results obtained with different values of k are mostly not complementary, thus different values of k cannot be used to generate competing portfolio engines.

Table 6.7 shows our results gathered combining the two best values of k of

	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8
McSplit	35	28	36	37	36	36	34	35
McSplit+RL	35	28	36	37	36	37	35	35

Table 6.6: Number of wins of our cumulative cosine similarity-based strategy over McSplit original heuristic and McSplit+RL on larger graphs.

both heuristics, namely $k = 1$ and $k = 2$ for the norm and $k = 3$ and $k = 4$ for the cumulative cosine similarity heuristic. The table shows how these strategies outperform the original heuristics in more than 90% of the cases.

	CCS k=3	CCS k=4	Norms k=1	Norms k=2
Wins	36	42	45	46
[%]	72	84	90	92

Table 6.7: Cumulative (from left to right) results when combining our heuristics with the 2 best values of k against McSplit.

6.2 Best-pair McSplit Implementation

In this section, we show and comment the tests performed to evaluate our best-pair McSplit implementation (Section 5.3). All tests were performed on the same hardware as in the previous section (i9 9900k + GTX 1060). In this case, the solver is fully implemented in python. In future works, if this method is improved, it is also possible to use the python script only to pre-compute embeddings, and load them in a C++ environment as we did with node heuristics. Of course, we compare our modified solver with a python re-implementation of McSplit. We test on 70 instances, randomly selected from McSplit original repository on GitHUb.

We can see in Table 6.8 that unfortunately, our methods cannot outperform McSplit. However, they can reach a relatively big solution size in average. This shows that prioritizing node pairs instead of single nodes, might still work with a different heuristic. At the moment, the used heuristic is to prioritize node pairs using cosine similarity. We can either improve our embeddings, so that, the cosine similarity leads to better results, or find another method to prioritize node pairs.

	BPBD	BP
Wins	3	5
Losses	67	65
Solution Size Ratio (AVG)	0.85	0.87

Table 6.8: Our two modified version of McSPlit (Best Pair on selected BiDomain (BPBD) and Best Pair among all bidomains (BP) compared with the original McSplit.

6.3 Custom Neural Network

Our custom neural network is still in a development phase. The current prototype has been presented in Section 5.4. We must first ensure that this network is capable of learning the similarities between nodes in two input graphs, so, we have been testing on a simpler problem. We try to teach the network to recognize nodes with the same degree. The network receives as an input 1 adjacency matrix and 1 feature vector for each graph. Each graph is processed by the first neural network. The resulting embeddings are fed together to the second neural network. As a ground truth, we have a matrix M that represents all the possible node pairings between one node of the first graph and one node of the second. The matrix encodes, with binary labels, if the two nodes have the same degree, (label 1), or they have different degree, (label 0). For instance, if M_{12} contains 1, then node 1 in the first graph has the same degree of node 2 in the second graph. From a data representation perspective, the new problem is totally equivalent to the MCS. The only difference, is that the ground truth matrix will represent another notion of similarity, that is, two nodes are similar if they are matched in the same MCS solution by McSplit, instead of having the same degree. For the training we use the Adam optimizer. We tune the hyperparameters in the validation step. In particular, the batch size will be selected between 32, 64 and 128 while the learning rate will be selected among 0.001, 0.001, 0.0001 and 0.00001. We used an embedding dimension of 1000. We are using either an L2 or L1 loss function (euclidean or Manhattan distance), but the final choice is still under discussion. Unfortunately, while the network is able to reach an high value of accuracy after training, this is due to the dataset being unbalanced since most nodes have not the same degree. Our network is

highly biased in predicting the 0 label, so its predictions have currently no value. We also expect this problem when switching to the MCS. Our latest efforts have been focused on balancing the dataset. We have two alternatives:

- Using a loss function that penalizes mistakes on label 1 more than mistakes on label 0.
- Generating a dataset with the same occurrences of label 1 and label 0 in the ground truth matrix.

In this case, since the generation of a balanced dataset is not an easy task, we are focusing the first alternative.

Chapter 7

Conclusions

In this chapter, we will resume our accomplishment and explain future development possibilities for each of the proposed methods. In section 7.1, we show future possibilities for our heuristics based on prioritizing nodes. In section 7.2, we talk about our modified best-pair McSplit, which has the potential of beating the original implementation, if a better way to prioritize node pairs is found. Finally, in section 7.3 we talk about how our custom neural network could be useful in finding the just mentioned node pair ordering for best-pair McSplit.

7.1 GNN-based heuristics

We proposed a branching heuristic based on Graph Machine learning. We used NeuroMatch to embed the k-hop neighborhood around each node to extract the order of the vertices to visit. Norm and cosine similarity are then used instead of node-degrees in prioritizing nodes when branching. We tried different values of the k-parameter and although with norms we couldn't gain a definitive advantage against the original heuristic, it perform better in about half of our test instances. We still have a clear advantage when considering the different orderings obtained with different values of k. Interestingly, the cumulative cosine similarity heuristic performs better for all values of k, and combining the best k values both in norms and cumulative cosine similarity we outperform the node-degree heuristic in almost all the instances. Among the future works we mention the following possibilities.

First of all, we would like to tune our approach to work with other kinds of graph pairs as well as finding a substitute for the McSplit bidomain heuristic, which limits our choices when branching. Indeed, NeuroMatch can be adapted to work with labeled, directed, and larger graphs. Moreover, we would like to verify the behavior of some supervised methods to perform the same task. Finally, we would like to run extensive experiments on large graphs to better evaluate the scalability of the new approach.

7.2 Best-pair McSplit Implementation

We proposed a new implementation of McSplit which no longer picks single nodes but prioritizes node pairs directly. This methodology has indeed potential since the MCS problem consists in finding node pairs that builds a matching. Giving priorities to nodes limit us to choose a node with an higher a priori possibility to be part of the MCS. Our implementation is fully functional but it needs a better heuristic. We proposed an heuristic based on cosine similarity, i.e., node pairs with an higher cosine similarity are select first. However, results were not as good as expected. In future works, we will try either to improve our embeddings quality, leading to a cosine similarity measure that is more informative and to find totally new heuristics for the node pair ordering that may or may not be based on Graph Neural Networks.

7.3 Custom Neural Network

This methodology has potential to be combined with the best pair McSplit implementation. Our custom neural network is probably our best bet on finding a good node pair ordering. We propose to compute embeddings using a different notion of similarity. Since a graph structure is what matters when matching nodes, all our previous attepts have been focused in learning embeddings directly from graphs, in an unsupervised or semi-supervised way. With our custom neural network we add a supervision, i.e., the original McSplit solution. McSplit solution contains a set of node pair matchings, we consider nodes in these matches to be similar. What we want is an neural network that maps node pairs into an embedding space where

this notion of similarity is respected. Much work is underway, we have been testing with a conceptually similar problem and we are starting to see some results. Our current concern is balancing our dataset, since our predictions are highly biased.

Bibliography

[1] Jonathan L Gross, Jay Yellen, and Mark Anderson. *Graph theory and its applications*. Chapman and Hall/CRC, 2018 (cit. on p. 2).

[2] Roberto Tadei and Federico Della Croce. *Elementi di ricerca operativa*. Societa' editrice Esculapio, 2019 (cit. on p. 4).

[3] Viggo Kann. «On the approximability of the maximum common subgraph problem». In: *STACS 92*. Springer Berlin Heidelberg, 1992, pp. 375–388. DOI: 10.1007/3-540-55210-3_198. URL: https://doi.org/10.1007/3-540-55210-3_198 (cit. on p. 7).

[4] Philippe Vismara and Benoıt Valery. «Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms». In: *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2008, pp. 358–368. DOI: 10.1007/978-3-540-87477-5_39. URL: https://doi.org/10.1007/978-3-540-87477-5_39 (cit. on p. 9).

[5] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. «Clique and Constraint Models for Maximum Common (Connected) Subgraph Problems». In: *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 350–368. DOI: 10.1007/978-3-319-44953-1_23. URL: https://doi.org/10.1007/978-3-319-44953-1_23 (cit. on p. 9).

[6] Ciaran McCreesh, Patrick Prosser, and James Trimble. «A Partitioning Algorithm for Maximum Common Subgraph Problems». In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 2017, pp. 712–719. DOI: 10.24963/ijcai.2017/99. URL: https://doi.org/10.24963/ijcai.2017/99 (cit. on pp. 9, 11).

[7] G. Levi. «A note on the derivation of maximal common subgraphs of two directed or undirected graphs». In: *Calcolo* 9.4 (Dec. 1973), pp. 341–352. DOI: 10.1007/bf02575586. URL: https://doi.org/10.1007/bf02575586 (cit. on p. 9).

[8] «Reinforcement Learning:» in: *Kybernetes* 27.9 (Dec. 1998), pp. 1093–1096. DOI: 10.1108/k.1998.27.9.1093.3. URL: https://doi.org/10.1108/k.1998.27.9.1093.3 (cit. on p. 16).

[9] Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. «A Learning Based Branch and Bound for Maximum Common Subgraph Related Problems». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.03 (Apr. 2020), pp. 2392–2399. DOI: 10.1609/aaai.v34i03.5619. URL: https://doi.org/10.1609/aaai.v34i03.5619 (cit. on pp. 16, 55).

[10] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994 (cit. on p. 20).

[11] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. «Weisfeiler-lehman graph kernels.» In: *Journal of Machine Learning Research* 12.9 (2011) (cit. on p. 23).

[12] Aditya Grover and Jure Leskovec. «node2vec: Scalable feature learning for networks». In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864 (cit. on p. 24).

[13] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. «Deepwalk: Online learning of social representations». In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 701–710 (cit. on p. 26).

[14] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. «Graph convolutional networks: a comprehensive review». In: *Computational Social Networks* 6.1 (2019), pp. 1–23 (cit. on pp. 28, 30).

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. «Inductive representation learning on large graphs». In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 30).

[16] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. «Graph attention networks». In: *arXiv preprint arXiv:1710.10903* (2017) (cit. on p. 31).

[17] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. «Understanding batch normalization». In: *Advances in neural information processing systems* 31 (2018) (cit. on p. 31).

[18] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: a simple way to prevent neural networks from overfitting». In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958 (cit. on p. 31).

[19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. «How powerful are graph neural networks?» In: *arXiv preprint arXiv:1810.00826* (2018) (cit. on p. 36).

[20] Rex, Ying, Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, and Jure Leskovec. *Neural Subgraph Matching*. 2020. arXiv: 2007.03092 [cs.LG] (cit. on pp. 39, 55).

[21] Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. «GLSearch: Maximum Common Subgraph Detection via Learning to Search». In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 588–598. URL: https://proceedings.mlr.press/v139/bai21e.html (cit. on p. 46).

[22] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. *Measuring and Relieving the Over-smoothing Problem for Graph Neural Networks from the Topological View*. 2019. arXiv: 1909.03211 [cs.LG] (cit. on p. 48).

[23] Stefano Quer, Andrea Marcelli, and Giovanni Squillero. «The Maximum Common Subgraph Problem: A Parallel and Multi-Engine Approach». In: *Computation* 8 (May 2020), p. 48. DOI: 10.3390/computation8020048 (cit. on p. 56).