Politecnico di Torino

POLITECNICO DI TORINO

Department of Control and Computer Engineering

MASTER'S DEGREE IN COMPUTER ENGINEERING

Academic Year 2021/2022

EFFECTIVE HEURISTICS FOR ASSESSING THE SIMILARITY OF A SET OF GRAPHS

Supervisors
Prof. Giovanni SQUILLERO
Prof. Stefano QUER

Candidate
Enrico CARRARO

July, 2022

ABSTRACT

The maximum common subgraph problem has been proven to have NP-hard complexity. Despite this theoretical limit, state-of-the-art algorithms made it viable to solve the problem exactly for some applications. This thesis discusses improvements to the state-of-the-art (a branch and bound algorithm with a smart partitioning strategy) in two areas: the generalization of the algorithm to handle sets of graphs and a best-first search space exploration strategy. The original algorithm is not designed to serve applications where time is constrained, and approximate solutions can be accepted, as it explores the solution space exhaustively. The newly proposed algorithm aims to achieve better approximate solutions in time-bounded problem instances. Tests demonstrated that the new method is scalable to multiple threads with low overhead. Finally, they can handle heterogeneous sets of graphs in sensible ways, i.e., discarding graphs that are not similar to most of the set. Graph Neural Networks are then used in other efforts to improve upon the heuristics for the state-of-the-art solver. Additionally, a variant of the original solver that employs heuristics to select node pairs as opposed to single nodes is discussed. Experiments demonstrated a significant improvement over the original method. A variant of the solver that employs heuristics to select node pairs rather than single nodes is also proposed.

ACKNOWLEDGEMENTS

"The dwarf sees farther than the giant, when he has the giant's shoulder to mount on."
The Friend, Samuel Taylor Coleridge

Without the insights and help from Thomas Madeo, Lorenzo Cardone and Andrea Calabrese this thesis would not have been what it is, thank you.

I thank my parents, grandparents, and my sisters Lisa and Sofia for their constant support throughout the years.

Finally, I thank Paola Vottero, that, over the past five years, helped and motivated me to become a better student and human being.

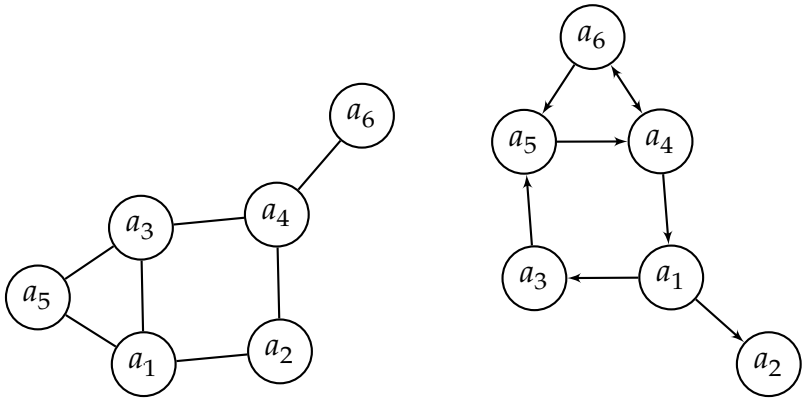
CONTENTS

1

INTRODUCTION

Graphs are structures made of a set of *vertices* and *edges*, which are pairs of adjacent vertices. Graphs are often drawn using rings or dots to represent vertices and lines for edges. *Labels*, *weights*, or other data may be associated with vertices or edges. Sometimes edges are directed, in which case they are drawn as arrows rather than lines.

Thanks to their intuitive, machine-readable representation, graphs are a foundational data structure in a variety of applications. They are used to represent 3D objects, task dependencies, molecules, source code, executable binaries, social networks and many other types of data [59, 26].



(a) Undirected graph. (b) Directed graph.

Figure 1: Examples of graphs structures.

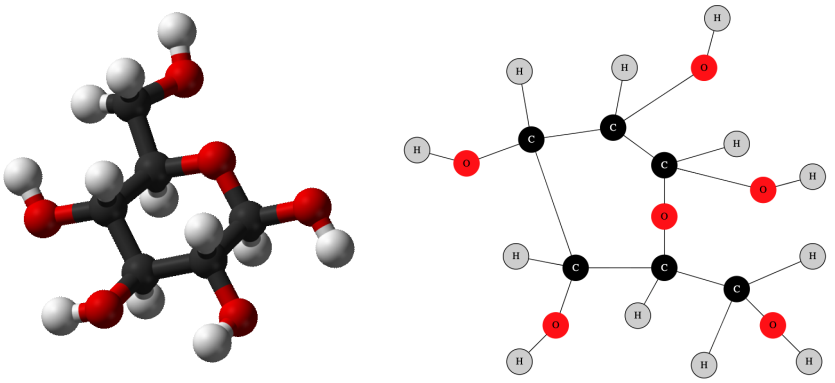
1.1 GRAPH SIMILARITY

In many computer vision and pattern recognition applications, where graph representations are frequently employed to characterize objects or interactions between elements, assessing similarity between graphs is crucial. However, selecting a similarity notion is not straightforward and can be greatly influenced by the application at hand.

This section contains some examples of graph similarity application followed by an overview of pairwise graph similarity notions.

1.1.1 *Examples of graph similarity applications*

A common problem in *chemoinformatics* is defined as follows: given a large set of chemical compounds (represented as node- and edge-labeled graphs) having a specific *function* (e.g., toxicity) and another set of molecules that do not, the task is predicting whether an unknown molecule will exhibit this function. In this problem, it is a common assumption that molecules with similar structures have similar functional properties.



(a) Ball-and-stick model. [5]. (b) Graph representation.

Figure 2: Chemical structure and graph representation of beta-D-Glucose.

Another useful application of graph similarity is *computer vision*, in which locating "most of" a pattern graph within a target graph approximates a visual match. An example of this is well illustrated in image 3 from the 2017 paper on graph based object search [67].

Finally, graph similarity can be applied in the field of cluster analysis [16]. Cluster analysis, or more simply *clustering*, is applied in various fields including pattern recognition, information retrieval [42], bioinformatics, data compression, and machine learning. The goal of clustering is to organize a collection of items so that items in the same group (referred to as a cluster) resemble one another more closely than those in other clusters.

Homogeneity measures how much the objects in a cluster are similar. It is usually defined using the Shannon's entropy [57],

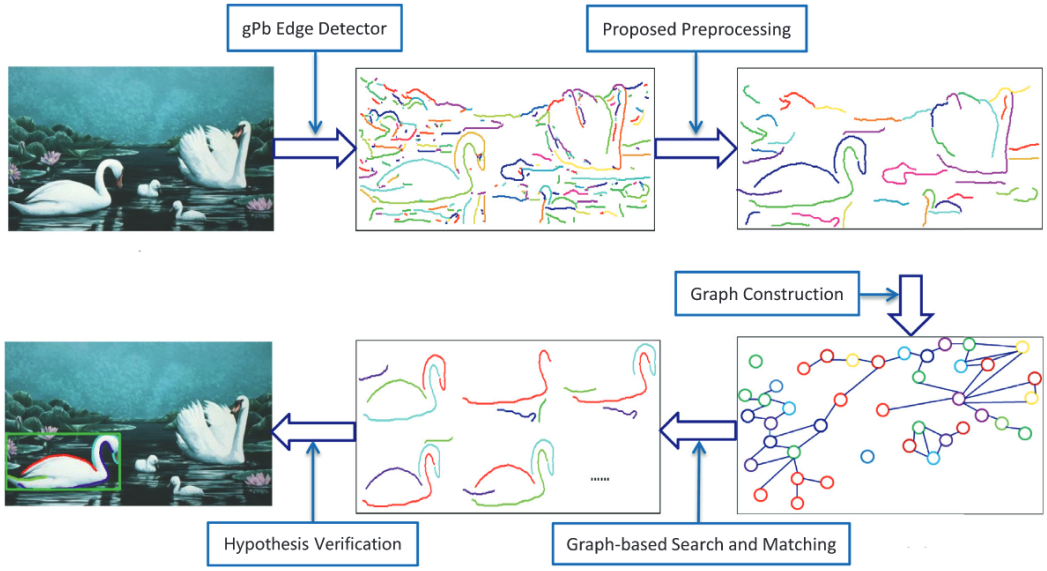


Figure 3: Graph based object detection pipeline by Wei et al. [67]

but when the objects are graphs it could be useful to define it as the graph similarity of the group.

1.1.2 *Graph similarity notions*

The definition of graph similarity can change depending on the use-case. This section contains a brief description of the most notable graph similarity notions.

Graph isomorphism is a binary similarity measure. A group of graphs is isomorphic if the graphs are topologically identical.

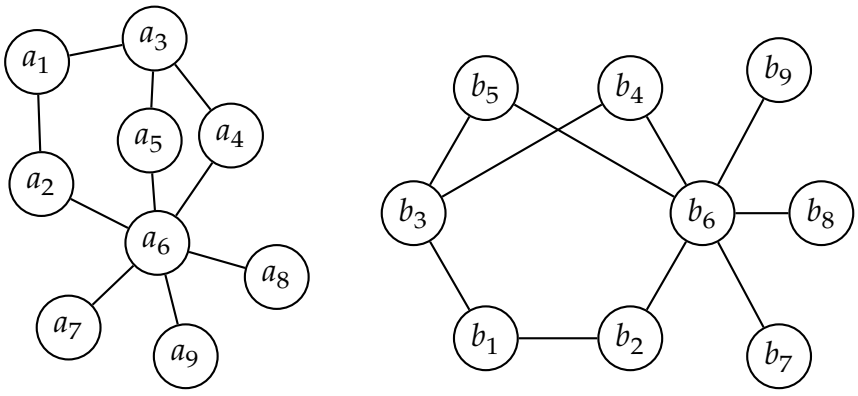
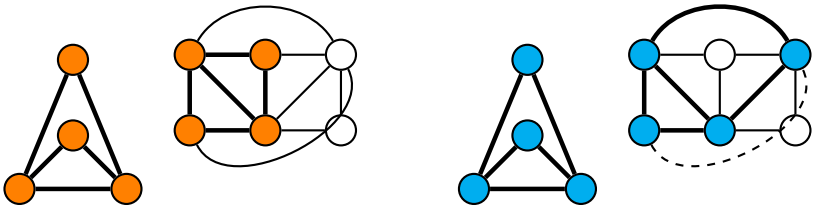


Figure 4: Example of structurally identical graphs.

An example of isomorphic graphs shown in figure 4.

Subgraph isomorphism is a generalized version of the graph isomorphism problem, it consists in looking for topologically identical copies of a graph G within another graph H.



(a) Example of induced subgraph isomorphism.

(b) Not an induced subgraph isomorphism due to the dashed edge.

Figure 5: Induced and non-induced subgraph isomorphism.

As seen in Figure 7a and 7b, the problem exists in both induced and non-induced forms.

The *maximum common subgraph* of two graphs is the largest graph (in terms of nodes and edges) contained both graphs.

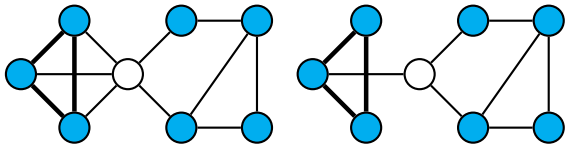


Figure 6: Example of maximum common subgraph.

The *minimum common supergraph* is the smallest graph that contains the subject graphs.

The minimum number of operations (node and edge additions, substitutions, and deletions) required to transform one graph into the another is the *error-correcting graph matching* (or *edit distance*). This is a generalization of the isomorphism problem, which asserts that two graphs are isomorphic if their edit distance is zero.

While the graph edit distance is a very broad notion, that can be used to derive both the maximum common subgraph and graph isomorphism, it is inherently a binary operator, and thus cannot be applied to a set of graphs. Conversely, the maximum common subgraph problem, can be applied to multiple graphs at once; this is why the present work focuses on heuristics and algorithms for the maximum common subgraph problem and its variations.

1.2 FORMAL NOTATION AND DEFINITIONS

The following is a list of fundamental definitions from graph theory [6] to fully comprehend the present work.

Definition 1.2.1 *A **graph** G is an ordered pair of disjoint sets (V, E) such that E is a subset of the set $V^{(2)}$ of pairs of V. The set V is the set of vertices, and E is the set of edges.*

*If G is a **graph**, then $V = V_G$ is the vertex set of G, and $E = E_G$ is the edge set.*

In this thesis, only finite graphs are considered; that is, V and E are always finite.

Definition 1.2.2 *The **order** of G is the number of vertices in G; it is denoted by $|G|$. The same notation is used for the cardinality, i.e. the number of elements of a set: $|X|$ denotes the number of set elements X. Thus $|G| = |V_G|$.*

*The **size** of G is the number of edges in G; it is denoted by E_G. G^n is the notation for an arbitrary graph of order n. Similarly, $G(n, m)$ denotes an arbitrary graph of order n and size m.*

Definition 1.2.3 *An edge $\{x, y\}$ is said to join the vertices x and y and is denoted by xy. If the edges are ordered pairs of vertices, the notion of a **directed** graph is introduced; otherwise, the graph is said to be **undirected**; thus, for undirected graphs, xy and yx mean the same edge.*

Except where otherwise stated, graphs in this thesis are *undirected*.

Definition 1.2.4 *If $xy \in E_G$ and then x and y are **adjacent** vertices of G, and the vertices x and y are incident with the edges xy and yx. Two edges are adjacent if they have exactly one common endvertex.*

If G is a directed graph and $xy \in E_G$, then x is adjacent to y, but y is only adjacent to x if $yx \in E_G$.

Definition 1.2.5 *The set of vertices adjacent to a vertex $x \in G$, the **neighbourhood** of x, is denoted by $\Gamma(x)$. The **degree** of x is $d(x) = |\Gamma(x)|$.*

In order to emphasize that the underlying graph is G, the neighbourhood and degree are denoted by $\Gamma_G(x)$ and $d_G(x)$, respectively; a similar convention will be adopted for other functions depending on an underlying graph.

Definition 1.2.6 *The **adjacency matrix** $A = A(G) = (a_{ij})$ of a graph G is the $n \times n$ matrix given by*

$$a_{ij} = \begin{cases} 1 & \text{if } v_i v_j \in E_G, \\ 0 & \text{otherwise} \end{cases}$$

Definition 1.2.7 *For directed graphs, the **incidence matrix** $B = B(G) = (b_{ij})$ of G is the $n \times m$ matrix defined by*

$$b_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is the initial vertex of the edge } e_j, \\ -1 & \text{if } v_i \text{ is the terminal vertex of the edge } e_j, \\ 0 & \text{otherwise.} \end{cases}$$

Definition 1.2.8 *A **path** is a graph P of the form*

$$V(P) = \{x_0, x_1, ..., x_l\}, E(P) = \{x_0x_1, x_1x_2, ..., x_{l-1}x_l\}.$$

This path P is usually denoted by $x_0, x_1, ..., x_l$. The vertices x_0 and x_l are the end vertices of P and $l = e(P)$ is the length of P. P is defined as a path from x_0 to x_l, or an x_0-x_l path.

Definition 1.2.9 *A **walk** W in a graph is an alternating sequence of vertices and edges, say $x_0, e_1, x_1, e_2..., e_l, x_l$ where $e_i = x_{i-l}x_i, 0 \leq i \leq l$.*

*In accordance with the terminology above, W is an x_0-x_l walk denoted by $x_0, x_1, ..., x_l$; the length of W is l. This walk W is called a **trail** if all its edges are distinct. Note that a path is a walk with distinct vertices. A trail whose end vertices coincide (a closed trail) is called a **circuit**.*

Definition 1.2.10 *A graph is **connected** if for every pair $\{x, y\}$ of distinct vertices there is a path from x to y.*

Definition 1.2.11 *A graph without any cycles is a **forest**, or an **acyclic graph**; a **tree** is a connected forest.*

Definition 1.2.12 *$G' = (V', E')$ is defined as a **subgraph** of $G = (V, E)$ if $V' \subset V$ and $E \subset E$. In this case, we have that $G' \subset G$.*

Definition 1.2.13 (Induced subgraph) *If G' contains all edges of G that join two vertices in V' then G' is said to be the **subgraph induced** or **spanned** by V' and is denoted by $G[V']$.*

Definition 1.2.14 *Thus $G = (V, E)$ is **isomorphic** to $G' = (V', E')$ if there is a bijection $\phi : V \to V'$ such that $xy \in E$ iff $\phi(x)\phi(y) \in E'$. Clearly, isomorphic graphs have the same order and size. Two isomorphic graphs G and H can be denoted by either $G \cong H$ or simply $G = H$.*

Definition 1.2.15 *A graph of order n and size m is called a* ***complete n-graph*** *and is denoted by* K_n.

Definition 1.2.16 *A maximal complete subgraph of a graph is a* ***clique****.*

Definition 1.2.17 *A graph is* ***labeled****, when information, such as labels or weights, characterizes its vertices or edges.*

1.3 THESIS OVERVIEW

This work proposes new strategies to tackle the maximum common subgraph (MCS) problem. It also uses a variation of the MCS problem applied to a set of graphs to evaluate their overall similarity.

The following is an overview of the organization of this work:

CHAPTER 1 Introduction to graph similarity, its applications and formal notation.

CHAPTER 2 Overview of pair-wise and group graph similarity.

CHAPTER 3 Survey of various approaches used to tackle the maximum common subgraph problem.

CHAPTER 4 Smart solution space exploration using best-first heuristics for the multi-graph MCS problem.

CHAPTER 5 GNN-based heuristics for the MCS.

CHAPTER 6 Conclusions.

2

GRAPH SIMILARITY NOTIONS IN DETAIL

Let $\Xi(\mathcal{G})$ be the similarity of a set of graphs

$$\mathcal{G} = \{G_1, G_2, ..., G_n\}$$

where the cardinality of the set is at least 2, i.e. $n = |\mathcal{G}| \geq 2$.

This section explores in detail the most important measures that can be used as $\Xi(\cdot)$ to assess the similarity of group of graphs.

Graph isomorphism

Possibly the most natural method to measure the similarity of graphs is to check whether they are topologically identical, that is, isomorphic. This gives rise to a binary similarity measure.

Despite the idea of checking graph isomorphism being so intuitive, no efficient general algorithms are known for it.

The complexity of determining whether two graphs are isomorphic is still unknown. Polynomial-time algorithms have been developed for several special classes of graphs, including trees and planar graphs [63], but many existing algorithms, which often seek to find the correct matrix permutation, are exhaustive in the worst case: see, for example, [64]. It is currently postulated that graph isomorphism may lie strictly between the $\mathcal{P}$ and $\mathcal{NP}$-complete complexity classes [43], i.e. $\mathcal{NP}$-intermediate.

An example of isomorphic graphs is shown in figure 4.

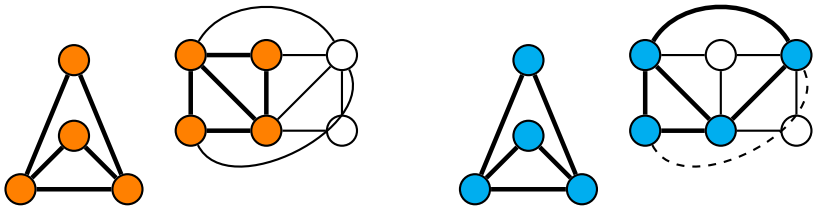
If graphs G and H are isomorphic, then a permutation matrix P exists, that will transform the adjacency matrix A_G into the adjacency matrix A_H, i.e. $A_H = PA_GP^T$. Additionally, if G and H are attributed graphs, then an isomorphism between them must also preserve the attributes on each node and edge.

Using graph isomorphism to define graph similarity leads to the following definition:

$$\Xi(\mathcal{G}) = \begin{cases} 1 & \text{if } \forall G_i \in \mathcal{G}, G_i \text{ is isomorphic to } G_{i+1}, \\ 0 & \text{otherwise.} \end{cases}$$

Subgraph isomorphism

A generalization of the graph isomorphism problem is that of *subgraph* isomorphism, in which one looks for isomorphic copies of a graph G within another graph H.



(a) Example of induced subgraph isomorphism.

(b) Not an induced subgraph isomorphism due to the dashed edge.

Figure 7: Induced and non-induced subgraph isomorphism.

Unlike the graph isomorphism problem, the problem of subgraph isomorphism has been proven to be $\mathcal{NP}$-complete [19, Section 3.2.1].

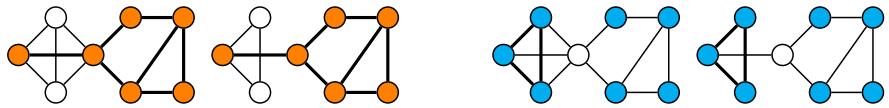
As seen in Figure 7a and 7b, the problem exists in both induced and non-induced forms.

Subgraph isomorphism does not lend itself well to be used as a similarity measure, as it can only assert if a graph (pattern) is present into another graph (target).

Finally, if the pattern does not appear, one may want to know how much of it can be found; this is where the maximum common subgraph (MCS) problem comes into play.

Maximum common subgraph

The maximum common subgraph of two graphs is the largest graph (in terms of nodes and edges) that is present in both graphs. The problem is $\mathcal{NP}$-complete [19, Section 3.3].



(a) Example of connected maximum common subgraph.

(b) Example of maximum common subgraph.

Figure 8: Maximum common subgraphs.

Maximum common subgraph problems are crucial in comparing graphs: in order to figure out what two graphs don't

have in common, one must first determine what they have in common[27]. They have been widely applied in biology and chemistry, computer vision, source code and binary programs analysis, circuit designs, computer-aided manufacturing, crisis management, dataset deanonymization, and character recognition problems; the biochemical applications alone are significant enough to justify extensive research into the problem [18]. The case for multiple graphs, as described by Hariharan et al. in 2011 [23], is mainly used in molecular science, and many of the currently used methods are based on choosing a substructure that is present in one molecule or that is shared by two, and then confirming that it is present in the other molecules.

The MCS problem comes in many forms; several of the applications listed above, including biochemistry, require the MCS to be connected. Figures 8a and 8b illustrate the difference that constraining the MCS to be connected makes.

Similarly to subgraph isomorphism, there are also induced and non-induced version of the problem. The goal of calculating the maximum common induced subgraph is to find a graph with as many vertices as possible that is an induced subgraph of each of two input graphs. It is necessary to map non-edges to non-edges and edges to edges. The aim of solving the maximum common partial subgraph problem (non-induced form) is to find a subgraph with as many edges as possible (vertices must be matched to vertices, but there may be additional edges in the solution).

Using the MCS to derive a graph similarity measure is intuitive, but given the many variants the MCS problem has, different $\Xi(\mathcal{G})(\cdot)$ can be derived. When computing the MCS for a pair of graphs, i.e. $|\mathcal{G}| = 2$, we can Let $\mathfrak{G}_\mathfrak{I}$ be the maximum common induced subgraph of $\mathcal{G}$, then $\Xi(\mathcal{G}) = |V(\mathfrak{G}_\mathfrak{I})|$ for the induced version of the problem.

The number of vertices and edges of the resulting graphs should be taken into account when thinking about the maximum common partial subgraph problem, with the two factors possibly being weighted. Being $\mathfrak{G}_\mathfrak{P}$ the maximum common partial subgraph of $\mathcal{G}$, then

$$\Xi(\mathcal{G}) = \alpha \cdot |V(\mathfrak{G}_\mathfrak{P})| + \beta \cdot |E(\mathfrak{G}_\mathfrak{P})|$$

When dealing with more than one graph, the MCS problem can be relaxed to have some vertices attributed only to a subset of $\mathcal{G}$, meaning that the MCS is made up of vertices that are common among all of $\mathcal{G}$ and some that can be matched only to a subset of $\mathcal{G}$. For the sake of simplicity, the induced version of the problem is considered here. We have that:

$$\Xi(\mathcal{G}) = \sum_{i=1}^{|\mathcal{Q}|} \xi^{Q_i} \tag{1}$$

where $\xi \geq 1$ affects the balance between having many equivalence sets and having equivalence sets with many vertices (small ξ vs. large ξ). While $\mathcal{Q}$ is a set of equivalence sets

$$\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\} \tag{2}$$

such that each $Q_i \in \mathcal{Q}$ is a set of vertices

$$Q_i = \{v_{i,1}^{G_1}, v_{i,2}^{G_2}, ..., v_{i,m}^{G_m}\} \tag{3}$$

with the vertex $v_{i,j}^{G_j}$ belonging to graph G_j (i.e., $v_{i,j}^{G_j} \in V(G_j)$) and all vertices in Q_i belonging to different graphs (i.e., $\forall v_{i,j}^{G_j}, v_{i,k}^{G_k} \in Q_i : j \neq k \Rightarrow g_j \neq g_k$). Vertices belonging to the same equivalence set Q_i are called equivalent and denoted with $v_{i,j}^{g} \approx v_{i,k}^{h}$. If two vertices of the same graph G^g appearing in two different sets Q_i and Q_j are connected by an edge, then all vertices of Q_i and Q_j must be connected in their relative graphs, a requirement similar to the the one in induced equivalence of sub-graphs. More formally:

$$\begin{aligned} &\forall v_{i,k}^{G}, v_{i,l}^{H} \in Q_i \text{ and} \\ v_{j,m}^{G}, v_{j,n}^{H} \in Q_j : \left(v_{i,k}^{G}, v_{j,m}^{G}\right) \in E(G) &\Rightarrow \left(v_{i,l}^{H}, v_{j,n}^{H}\right) \in E(H) \end{aligned} \tag{4}$$

Minimum common supergraph

Like the maximum common subgraph, this graph similarity measure compares a set of input graphs by generating another.

The minimum common supergraph is the smallest graph containing the subject graphs. As observed in [8], algorithms for the maximum common subgraph can be easily modified to find the minimum common supergraph and vice versa. Note that the MCS of the set of graphs will be a isomorphic subgraph of the minimum common supergraph. This notion is trivially

applicable to a set of graphs, but it is easier to translate it into a dissimilarity measure because a very dissimilar set of graphs will lead to a large minimum common supergraph. Let $\mathfrak{g}$ be the minimum common supergraph of $\mathcal{G}$, then

$$\overline{\Xi(\cdot)} = |V(\mathfrak{g})|.$$

Graph edit distance

Image representations as graphs are frequently affected by missing or corrupted information in pattern recognition problems. Whole graph isomorphism will fail to recognize the images' underlying similarity in this case. Using an error-correcting procedure is one way to deal with corrupted data.

The minimum number of edit operations (node and edge additions, substitutions, and deletions) required to transform one graph into another is the error-correcting graph matching, or edit distance, between two graphs. This problem is a generalization of isomorphism, which states that two graphs are isomorphic if their edit distance is zero.

Typically, the costs of various edit operations are inversely related to the likelihood of the edit occurring. As a result, determining the edit distance between two graphs becomes an optimization problem: find the least expensive set of edit operations to transform one graph into another. Furthermore, if the nodes and edges of the graph have their attributes, different pairs of substitutions may have different costs. The cost function has a strong impact on the optimal error-correcting graph matching.

It can be proved that graph edit distance computation is equivalent to the maximum common subgraph problem when a particular cost function is used [7] on these nodes and edges. The graph edit distance between two attributed graphs, GED, G and G' is defined as:

$$\mathrm{GED}\left(G, G'\right) = \min_{\forall (e_1, e_2, \ldots e_k) \in E(G, G')} \left\{ \begin{array}{c} \mathrm{CED}\left(G, G'\right) \\ (e_1, e_2, \ldots e_k) \end{array} \right\}$$

where $E\left(G, G'\right)$ denotes the set of all edit paths $(e_1, e_2, \ldots e_k)$ that transform G into G' and the cost of the edit path is CED:

$$\begin{aligned} \mathrm{CED}\left(G, G'\right) = & \sum_{(e_1, e_2, \ldots e_k)} C_{\mathrm{vs}}\left(e_t\right) + \sum_{\forall e_t \in \mathrm{vs}} C_{\mathrm{es}}\left(e_t\right) + \sum_{\forall e_t \in \mathrm{vd}} C_{\mathrm{vd}}\left(e_t\right) \\ & + \sum_{\forall e_t \in ed} C_{\mathrm{ed}}\left(e_t\right) + \sum_{\forall e_t \in vi} C_{\mathrm{vi}}\left(e_t\right) + \sum_{\forall e_t \in ei} C_{\mathrm{ei}}\left(e_t\right) \end{aligned}$$

The set of node substitutions, deletions and insertions is represented by *vs*, *vd*, and *vi*, respectively. The costs of these edit operations are represented by C_{vs}, C_{vd} and C_{vi}. Edit operations and costs on edges are defined in a similar way: *es*, *ed*, *ei*, $C_{\text{es}}C_{\text{ed}}$ and C_{ei}.

A similarity measure can be simply derived using the graph edit distance but it can only be applied on pair of graphs. While the graph edit distance is a very broad and easily applicable notion of graph similarity, that can be used to derive both the maximum common subgraph and graph isomorphism, it is inherently a binary operator, and thus cannot be applied to a set of graphs.

Finding the graph edit distance is $\mathcal{NP}$-hard [70], and is even hard to approximate, being part of the $\mathcal{APX}$ class[14].

This work is focused on the maximum common subgraph problem because of the extensible nature of the problem and the vast literature on efficient algorithms to solve it.

2.1 HARDNESS

It is easy to fall into the trap of thinking that the problems above are easily solvable when presented with neat visual aids like coloring nodes and edges. Indeed it wouldn't be particularly challenging to solve the instances shown above by hand without any help; this is because the combinatorial nature of the problem is not visible in such examples, i.e., the order and size of the graphs is too small.

From complexity theory, we know that: problems where, if the answer is yes, then there is always a proof verifiable in polynomial time by a deterministic Turing machine; these problems are called non-deterministic polynomial, or $\mathcal{NP}$.

A property shared by MCS and other subgraph isomorphism problems is that they are $\mathcal{NP}$-complete. If an algorithm exists that can solve either of them in polynomial time, then an algorithm exists to solve any $\mathcal{NP}$ problem in polynomial time. Usually, a problem is determined to be $\mathcal{NP}$-complete by showing that an existing known $\mathcal{NP}$-complete problem can be reduced to it.

Currently it doesn't exist any algorithm capable of solving $\mathcal{NP}$-complete problems with polynomial time complexity.

3

APPROACHES TO THE MCS PROBLEM

This chapter begins with an introduction to constraint programming, describing the notions of *inference*, *search*, *heuristics* and *bounds*. The following sections are dedicated to the analysis of various resolution strategies that build on top of these fundamental concepts.

3.1 CONSTRAINTS MODELLING

Sudoku is probably the most well-known *constraint satisfaction problem* (*CSP*). Several variables can be accessed in a *CSP*, each of which can assume a specific range of values. In addition to this, a set of constraints denotes permitted combinations of values for a particular subset of the variables. To solve the game, one has to look for a way to assign a value from each variable's domain to satisfy all of the constraints on the problem. In order to find the optimal solution to a problem involving constraints and optimization, the original problem can be extended to consider various scoring functions [53].

The Sudoku game is set on a nine-by-nine grid. Some of the cells contain a number between one and nine, and the objective is to place a number between one and nine in each remaining box so that each number appears exactly once in each row, column, and each of the nine 3×3 subgrids. To represent this as a CSP, a variable is created for each of the $9 \times 9 = 81$ boxes, and each is assigned a domain consisting of the numbers one through nine. Then, a constraint is set on each row, column, and subgrid, stating that each of the nine variables must be unique. Since there are nine values and nine variables, *use each value only once* and *each value must be unique* are equivalent. Finally, for each pre-filled cell, there is a constraint stating that the corresponding box's value must match the specified value. Note that "proper" Sudoku puzzles are expected to have exactly one solution, whereas a CSP may have multiple solutions or none.

3.1.1 *Constraint Resolution*

One method for solving a CSP is to first generate every possible assignment of values to variables in turn and then check to see if all the constraints are satisfied. However, in most cases, the number of possible permutations renders such an approach impractical; consequently, algorithms with a higher level of intelligence are needed to deal with non-trivial issues. In this chapter, several different algorithms are examined, the majority centered on inference, bounds, and heuristic-driven search.

3.1.2 *Inference*

It is sometimes possible to deduce that certain variables can never be assigned a particular value. Since there is no point in generating any combination that contains a prohibited assignment, this line of reasoning can be used to reduce the amount of work needed. To represent this information algorithmically, one can keep track of the remaining values in the variable's domain, for each variable. Then, any value ruled out by the deductions is eliminated, much like when the remaining possible numbers in a Sudoku box are written and then crossed out as facts are deduced. A *propagator* is an algorithm that eliminates infeasible values.

Occasionally, a variable will have only one remaining value in its domain. This situation is commonly used to eliminate additional values; for instance, in *subgraph isomorphism problems*, if it is known that a particular pattern vertex v can only be mapped to a single target vertex w, any value representing target vertices that are not adjacent to v for each variable corresponding to a vertex adjacent to v can be eliminated. This can potentially have a cascade effect, allowing further systematic deletions [49]. Additionally, it may be possible to reason with combinations of domains with more than one remaining value. Figure 9 illustrates the specific case of graph isomorphism.

Sometimes it is viable to calculate higher levels of consistency [35, 54]. In some problems, for example, additional information can be inferred by checking support for each value v in a domain, i.e., looking at each constraint in turn that involves v's variable and verifying that at least one way of giving values to the other variables involved in that constraint that is consistent with v exists. A set of domains is said to be *arc consistent* (or generalized arc consistent if constraints involve more than two variables) if every value in the set is supported this way.

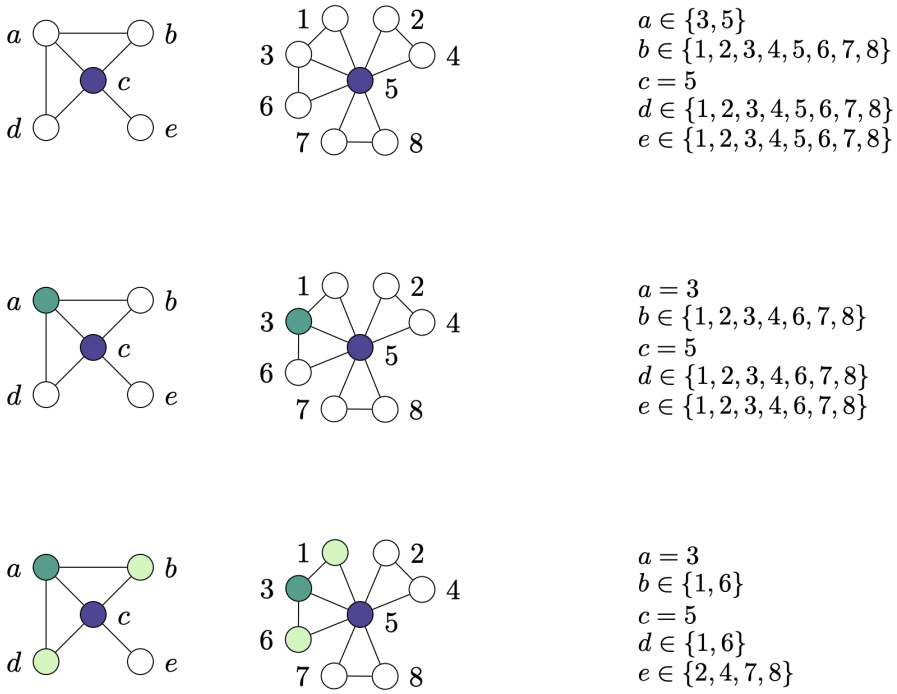


Figure 9: The top left vertex a in the pattern can only be mapped to pattern vertices 3 or 5, and the central vertex c can only be mapped to the target's central vertex 5. Since $c = 5$ is forced, no other pattern vertex can take 5; a's domain is 3. Consequently, 3 is removed from each domain. Because adjacent vertices must be mapped to adjacent vertices, b and d can only go to 1 and 6. There are no more forced assignments, but because b and d only have 1 and 6 between them, e cannot take either value [36]

.

3.1.3 *Exploration*

Despite the many different inference methods available, filtering is frequently insufficient to either locate a solution or prove that none exists. In this predicament, speculation is required: a variable is chosen and instructed to take one of the values still available within its domain. It is expected that this assignment will allow for further propagation. If this step alone does not succeed in locating a solution, the process is repeated, leading to a recursive search. Thus, at some point, the domain of a variable might become empty, meaning that a mistake was made (or that there is no solution). In such a situation, one must go back over the steps and try another value for the most recently guessed variable. When branching this way, there is also the possibility of running out of admissible values for one of the variables. If

this happens, one of the previously guessed variables needs to be restored; if this situation occurs for the first variable, it is proof that there is no solution.

Forward checking is the process of interleaving search with the basic propagation of constraints across domains that was just described. The expression *maintaining arc consistency* is used if the algorithm successfully keeps the arc consistent throughout each level of the search. *Conventional backtracking* refers to simpler algorithms that do not store domains at all and do not detect the lack of a value that is available for a variable until an assignment is attempted. These types of backtracking algorithms are known to be more efficient. When search and inference are performed sequentially, the effectiveness of the inference step is of the utmost importance. Extensive research has been conducted to identify an appropriate and principled order to make inferences for different sets of constraints.

A fully general constraint propagation algorithm needs to be able to determine not only when it is necessary to propagate a constraint, but also when it is not necessary to look at a constraint in order to avoid wasting effort and achieve the best performance possible. This requires knowledge of whether individual propagators guarantee properties such as *idempotence* (does running the propagator twice consecutively always give the same results as running it just once?) and *monotonicity* (does the propagator guarantee that it will never eliminate fewer values when applied to reduced domains?), besides having estimates of the relative execution costs of various propagators. In literature [55, 56] there are different perspectives on the design of such algorithms.

3.1.4 *Heuristics*

When branching during the search process, the choice of which variable and value to guess first has a significantly affects the total amount of time spent searching. There are good general principles that can be applied to the process of selecting variables [22]. One such principle is to select the domain that has the fewest number of values left first. Another principle is to select whichever domain is the most constrained.

This kind of rule is known as a *heuristic*; the term refers to the fact that it does not guarantee that it will provide the best choice, but it does, empirically speaking, tend to provide good choices the majority of the time. Value selection can be more challenging

when discussing solutions to graph problems, we typically end up discussing the number of neighbors that a vertex possesses. Unfortunately, value selection heuristics for the maximum subgraph isomorphism problem are not particularly reliable. This is especially true when, as is frequently the case, a large number of vertices have the same number of neighbors.

3.1.5 *Bounds*

When dealing with optimization problems such as the maximum common subgraph, a bound function can provide an additional form of inference. Let's say we want to find the MCS between two graphs. A feasible solution, i.e., any subgraph isomorphism, can be found by using inference and search, and this solution is called the *incumbent*. After that, the search goes on, but this time the focus is on finding subgraphs that are significantly larger than the already discovered one.

In order to accomplish this, the existing solution is used in a specific kind of filtering that removes portions of the search space that can be proved not to have a more optimal solution. The general idea is that each time a solution is discovered, a new constraint is added that states that "the solution must be better than this new incumbent", and the search continues until there is no other better solution.

3.1.6 *Microstructure*

The microstructure encoding of a CSP [25] is a method for representing a problem instance as a graph in which a clique of a specific size corresponds to a solution. In constraint programming, microstructure is primarily studied for its theoretical properties [10, 11, 13, 25], but it is sometimes also useful as a practical problem-solving technique.

3.1.7 *Smart versus Fast*

> *Complex methods to reduce the size of the search tree often do not lead to corresponding reductions in actual execution time, because of the additional work needed at each node.*
>
> Cook, Stephen A. and David G. Mitchell [12]

A trade-off is needed between the amount of propagation done and the amount of search that must be done. Even when

strong filtering is theoretically capable of removing more values, it is not always worthwhile to make every effort to remove values via propagation because expensive propagation algorithms may not result in additional deletions or, if they do, those deletions may not significantly reduce the amount of search necessary.

When choosing variable and value ordering heuristics, the proper balance must also be struck. For instance, it may be more profitable to select a vertex having the most neighbors at the beginning of the search rather than trying to find a vertex with the majority of neighbors still present in other variables (which must be calculated dynamically). Similarly, we can choose how much effort to put into obtaining very good bounds at the cost of increasing the cost of each step and how much effort to put into pruning symmetries.

3.2 CONSTRAING PROGRAMMING

Constraint programming (CP) is an established paradigm for solving combinatorial search problems. It incorporates approaches from artificial intelligence, operations research, algorithms, graph theory, and other disciplines. The core notion underlying constraint programming is that the user specifies the constraints, which are then solved using a general-purpose constraint solver.

Constraints are simply relationships, and a constraint satisfaction problem (CSP) specifies which relationships should exist between the decision variables [52].

The first explicit constraint programming model was created by Vismara and Valery in 2008 [66]. Given two graphs G and H, this model connects a variable D_v to every vertex v of G, and the domain of this variable includes all vertices of H plus an extra value $\perp$: If the vertex v doesn't match any vertex of H, variable D_v is given to $\perp$. If it does match a vertex of H, variable D_v is given to that vertex. Edge constraints make sure that variable assignments keep edges and non-edges between matched vertices:

$$\forall u, v \in \mathrm{V}(G), (i(u) = \perp) \vee (i(v) = \perp) \vee ((u, v) \in \mathrm{E}(G) \Leftrightarrow (i(u), i(v)) \in \mathrm{E}(H))$$

where $i(v)$ is the value that was assigned to the variable D_v. Difference constraints make sure that each vertex of H is assigned to no more than one variable, that is:

$$\forall u, v \in \mathrm{V}(G) \text{ distinct}, (i(u) = \perp) \vee (i(v) = \perp) \vee (i(u) \neq i(v)).$$

In 2011, Ndiaye and Solnon [41] enhanced this constraint programming model by exchanging binary difference constraints with a soft global all-different constraint. This change maximizes the number of D_u variables that are assigned to values that are distinct from $\perp$, while also ensuring that these variables are all distinct when they are not assigned to $\perp$. They concluded that forward checking achieves the best results on unlabeled graphs, while maintaining arc consistency [54] on the edge constraints achieves the best results on labelled graphs. Both methods are superior to those that came before them and achieve better results. This is a weaker version of global arc consistent soft all-different constraint by Petit, Régin, and Bessière [45] that is used in both cases. It uses a matching algorithm to check whether or not it is possible to assign distinct values to enough D_u variables to surpass the best cost found so far.

3.3 MCS VIA MAXIMAL CLIQUE

Solving the problem of finding the largest clique in an association graph can be an alternative approach to computing the maximum common subgraph [4, 17, 29, 48]. The association graph (also known as a compatibility graph or weak modular product) between two graphs G and H is an undirected graph $G \nabla H$ with vertex set

$$\mathrm{V}(G \nabla H) = \{(v, v') \in \mathrm{V}(G) \times \mathrm{V}(H) : (v, v) \in \mathrm{E}(G) \Leftrightarrow (v', v') \in \mathrm{E}(H)\}.$$

In order to avoid confusion between the vertices of $G \nabla H$ and the vertices of the two original graphs, the vertices of $G \nabla H$ are called *matching nodes*. This is because each vertex of (u, u') of $G \nabla H$ denotes the matching of u with u'. The edges of $G \nabla H$ connect matching nodes that denote compatible assignments. As a result, two matching nodes (u, u') and (v, v') are adjacent if $u \neq v$ and $u' \neq v'$, and if they preserve both edges and non-edges. As a result $(u, v) \in \mathrm{E}(G) \Leftrightarrow (u', v') \in \mathrm{E}(H)$. Figure 10 illustrates this concept.

A clique in an association graph represents a collection of compatible pairings. Consequently, such a clique corresponds to a common subgraph, and the maximum clique of $G \nabla H$ corresponds to the maximum common subgraph of G and H. Any method capable of locating the largest clique in a graph can therefore be used to solve the maximum common subgraph problem.

Note that the association graph is a subgraph of the microstructure [25] associated with Vismara and Valery's constraint pro-

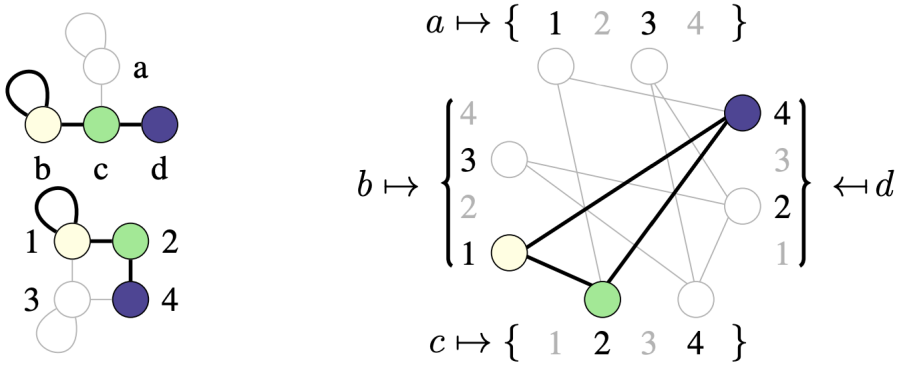


Figure 10: The two graphs on the left have a maximum common induced subgraph with three vertices. One solution is shown in bold. On the right is the association graph encoding: the highlighted three-member clique shows the same answer. The "missing" vertices do not have assignable edges due to the presence or absence of loops.

gramming model [66]; the microstructure has more matching nodes than the association graph because it contains a matching node $(u, \perp)$ for each vertex u of G. Each clique of size $|V(G)|$ corresponds to a subgraph whose size is defined by the number of matching nodes that do not contain $\perp$.

According to previous research [36], the clique-based algorithm outperforms constraint programming models with labelled graphs; however, when dealing with unlabelled graphs, both approaches perform similarly.

3.4 BRANCH AND BOUND

In 1973, Levi [29] published an algorithm to solve the MCS problem using the concept of *maximal compatibility classes*.

Later, McGregor [40] presented a branch-and-bound MCS algorithm with backtrack search. It was used as a program component for analysing chemical reactions and enumerating the bond changes that have taken place. This algorithm bounding function estimates the number of vertices that can still be matched, cutting the current branch as soon as its bound drops below the size of the largest known common subgraph; a mapping between two nodes of the input graphs is made for each branch.

Krissinel, Evgeny and Henrick [28] improved McGregor's algorithm with a more efficient backtrack search in 2004.

In 2017 McCreesh, Prosser and Trimble introduced *McSplit*, a new method based on vertex labelling and partitioning, that reliably outperforms traditional CP by an order of magnitude. It is the state-of-the-art algorithm to solve the *maximum common subgraph* and *maximum common connected subgraph* problems. It takes advantage of a CP invariant to reduce memory requirements and allow for more powerful and time-efficient branching strategies [37].

3.4.1 *McSplit*

McSplit [37] improves backtracking and pruning operations while retaining the branching and filtering benefits of constraint programming; it also achieves a consistent speedup of up to an order of magnitude for solution space exploration when compared to state-of-the-art in constraint programming.

Thanks to the sparing use of memory, it manages to handle larger graphs than its competitors. Thanks to the clever design of the graph data-structure it uses, the algorithm is able to solve the *maximum common partial subgraph* and the *maximum common induced subgraph problems* for *undirected*, *directed*, and *labeled* graphs.

General Description

Let G and H be two undirected and unlabelled graphs of order g and h, respectively. The goal is to find a mapping $M = \{(v_1, w_1), \dots, (v_n, w_n)\}$ of $|M| = m$ vertex pairs, where $v_i \in V(G)$ and $w_i \in V(H)$ are distinct vertices from the input graphs, such that v_i and v_j are adjacent in G if and only if w_i and w_j are adjacent in H.

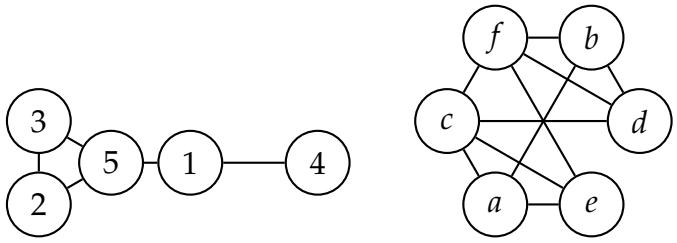
The subgraph in G induced by$\{v_1, \dots, v_n\}$ and the subgraph in H induced by $\{w_1, \dots, w_n\}$ are isomorphic by construction and correspond to the maximum common subgraph of G and H given the largest mapping, $|M^*| = \max(m)$.

The depth-first search-based algorithm has a recursive structure. Starting from $\emptyset$ at each depth level, for each vertex $v_i \in G$, it first attempts all matches (v_i, w_i) with all the $w_j \in H$;, computing any potential subgraphs while leaving the vertex v_i unmatched.

The graphs G and H in Figure 11 are useful to analyze a simple example. They share a four-vertex maximum common subgraph; one of the possible solutions is the mapping $\{(1, a), (2, f), (3, d), (5, b)\}$. The subgraph of G induced by vertices $1, 2, 3, 5$ is therefore isomorphic with the subgraph of H induced by the vertices a, b, d, f.

A fundamental aspect of the algorithm is the labeling of vertices as the search progresses. Every time a new pair is added to the mapping, all the other vertices are assigned a new label that keeps track of whether or not they are adjacent to every existing vertex. These labels are referred to as *adjacency classes*. The set of nodes belonging to a certain adjacency class is also called a *domain*. *Bidomain* refers to the set of nodes belonging to the same adjacency class when considering both input graphs, which can be stored in memory efficiently.

Returning to the example, the algorithm first arbitrarily selects vertex 1 from G and then attempts to match it with vertex a from H.



(a) Graph G. (b) Graph H.

Figure 11: Example graphs.

Vertex	Label
2	0
3	0
4	1
5	1

(a) Labelling of G.

Vertex	Label
b	1
c	1
d	0
e	1
f	0

(b) Labelling of H.

Figure 12: Node labels after mapping 1 to a.

Vertex	Label
3	01
4	10
5	11

(a) Labelling of G.

Vertex	Label
b	11
c	11
e	10
f	01

(b) Labelling of H.

Figure 13: Node labels after mapping 2 to d.

Vertex	Label
4	100
5	111

(a) Labelling of G.

Vertex	Label
b	111
c	111
e	101

(b) Labelling of H.

Figure 14: Node labels after mapping 3 to f.

As shown in the tables in Figure 12, each unmatched vertex in $V(G)$ is labeled based on whether it is adjacent to vertex 1, and each unmatched vertex in $V(H)$ is labeled based on whether it is adjacent to vertex a. Adjacent vertices have label 1; non-adjacent vertices have label 0.

The same strategy is applied for each subsequent recursion: the mapping M can be extended with a new pair (v, w) if and only if v and w share the same label. This property of mapping vertices with identical labels from the two input graphs is the algorithm's central feature.

The following step is to map another vertex of G to an unmatched vertex in H having the same label. Assuming the algorithm selects the vertices 2 and d, the mapping is now $M = \{(1, a), (2, d)\}$. Now, a two-character string is used to label unmatched vertices, with the first character being the label of the previous step and the second character indicating whether the vertex is adjacent to the newly mapped vertex or not. For instance, vertex 3 is labeled with 01 because it is not adjacent to vertex 1, but it is adjacent to vertex 2, which are already present in M. (Figure 13). Similar labels are assigned to non-mapped vertices in $V(H)$, indicating adjacency to mapped vertices a and d.

Only vertices that have the same label can be mapped together and added to the mapping in the subsequent steps, so the invariant method will continue to be maintained throughout these steps.

At each level, it is possible to determine the maximum number of possible pairs that can be added to the mapping before going back and attempting other mappings. In Figure 14, for example, four distinct labels are used: 100, 101, 011, and 111.Since it is not possible to identify a matching element in the other graph, new pairs formed with those vertices cannot be added to the mapping in the future. Since the first three only appear in one graph, it is impossible to add them to the mapping. Instead, the label 111 appears once in G and once in H, which means that it is possible

to generate one new pair of vertices with this label before having to resort to backtracking.

Therefore, the upper bound of the mapping size is the sum of the smallest number of occurrences for each label that occurs at least once in both graphs. The bound formula is

$$\begin{aligned} \text{bound } = |M| + \textstyle\sum_{l \in L} \min\, (&|\{v \in V_G : \text{label}(v) = l\}| \\ &|\{w \in V_H : \text{label}(w) = l\}|) \end{aligned} \tag{5}$$

where L represents the collection of all labels used in both graphs.

Analysis of the Algorithm

Label-classes, or groups of vertices with the same label, and a recursive function 1 with two parameters are the algorithm's defining characteristics. $future$, which contains the list of currently available label-classes, and M, which is the current mapping of vertices, are the function's parameters. During each recursion iteration, the optimal solution is used to update a global structure, the incumbent.

At each run of the function search, first, if a greater mapping has been found, the global incumbent is updated; then, the upper bound for the current search branch is computed, and if the bound has been reached (i.e., the bound is less than or equal to the size of the actual incumbent, so further searches cannot improve it), the branch is pruned, and the function returns (lines 3 and 4).

The actual exploration can now start. A label class is first chosen from $future$ using a predetermined heuristic, and then a vertex v belonging to that label class is chosen from the G space and excluded from further searches. An iteration is carried out on each vertex $w \in H$ that belongs to the same label class. The effects of including the pair (v, w) in the mapping for each of them is investigated. Considering (v, w) as a new pair in the mapping, each label-class is now split into two new classes, depending on whether the vertices belonging to it are adjacent to v and w. This process is repeated for all label-classes in $future$.

Vertices in G next to v and H next to w are found in the first class (lines 10 to 13). This callss is then merged with $future'$ if there is at least one vertex in both sets. The same is repeated for vertices that are not adjacent (lines 14 to 17). A recursive call to `Search` is performed with $future$ and $M \cup \{(v, w)\}$; consequently, a new search is conducted with one additional pair in the mapping and new label classes that account for this addition.

Algorithm 1 McSplit.

```
1  Search(future, M) begin
3      if |M| > |incumbent| then incumbent ← M
4      bound ← |M| + Σ_{⟨G,H⟩∈future} min(|G|, |H|)   if bound ≤
       |incumbent| then return
6      ⟨G, H⟩ ← SelectLabelClass(future)   v ← SelectVertex(G)
8      for w ∈ H do
9          future′ ← ∅   for ⟨G′, H′⟩ ∈ future do
11             G″ ← G′ ∩ N(G, v) {v}   H″ ← H′ ∩ N(H, w) {w}   if
               G″ ≠ ∅ and H″ ≠ ∅ then
14             future′ ← future′ ∪ {⟨G″, H″⟩}
15             G″ ← G′ ∩ N̄(G, v) {v}   H″ ← H′ ∩ N̄(H, w) {w}   if
               G″ ≠ ∅ and H″ ≠ ∅ then
18             future′ ← future′ ∪ {⟨G″, H″⟩}
19         Search(future′, M ∪ {(v, w)})
20     G′ ← G {v}   future ← future {⟨G, H⟩}   if G′ ≠ ∅ then future ←
       future ∪ {⟨G′, H⟩}
23     Search(future, M)
24 McSplit(G, H) begin
26     global incumbent ← ∅   Search({⟨V(G), V(H)⟩}, ∅)
28     return incumbent
```

After every possible pairing between v and the vertices in H with the same label has been investigated, a new recursion is performed considering solutions with v unmatched. v is removed from G, and if it was the final vertex to be considered in its label-class, its label-class is also removed from $future$.

3.5 MCSPLIT WITH REINFORCEMENT LEARNING

Reinforcement learning (RL) is a machine learning technique in which an agent is rewarded for desirable behavior and punished for undesirable behavior [50]. The agent interacts with the external environment, possesses a particular state, and generates an action. The external environment provides the agent with feedback on the performed action. The purpose of the reward or punishment is to improve the agent's behavior in the future. The agent essentially learns through trial and error, similar to humans.

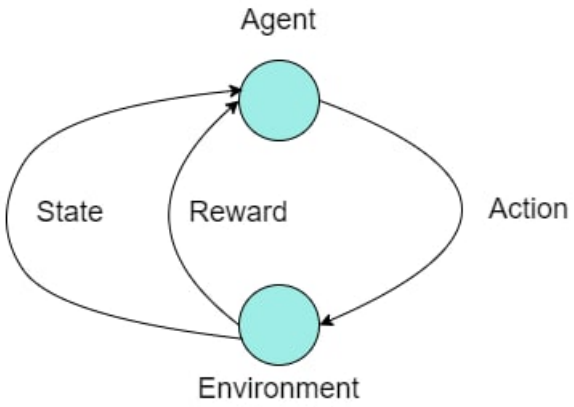


Figure 15: In a typical Reinforcement Learning scenario, an agent's actions in an environment are interpreted into a reward and a state representation, which are then fed back to the agent.

This framework was used to solve the MCS problem by incorporating an agent into the standard McSplit implementation. This new technique was coined as McSplit+RL [31]. The reinforcement learning agent is used to learn the optimal branching decisions, i.e., the branches that reduce the size of the search tree. Each branching option is considered an action; when the agent executes a branch, it receives a reward proportional to the search space reduction. The agent will choose the path with the greatest expected reward at each fork. McSplit always branches on the nodes with the highest degree, given a label class. McSplit+RL can branch on lower-degree nodes if the reinforcement learning agent anticipates a greater reward from them. McSplit+RL has been shown to perform better than McSplit, excluding instances where the reinforcement learning overhead is more computationally expensive than solving the problem itself. Both methods seek to identify the optimal MCS solution. McSplit+RL was able to solve approximately a hundred more problems within the predetermined time limit of 1800 seconds when tested on approximately 3000 problem instances.

3.6 PORTFOLIO APPROACH

While some algorithms are better on average than others, there is rarely a single best algorithm for a given problem. Instead, it is common for different algorithms to perform well on differ-

ent problem instances. When algorithms have a lot of run time variance, it can be difficult to decide which one to use. This particular problem has been called *algorithm selection problem* by Rice in 1976 [51]. Since run times for $\mathcal{NP}$-Hard algorithms are often highly variable from instance to instance, this phenomenon is most pronounced for these algorithms [30]. One can take advantage of such differences by combining several algorithms into a portfolio.

An example of a method using the portfolio approach is the *Glasgow subgraph solver* [39], which combines constraint programming concepts with a variety of powerful but fast domain-specific search and inference techniques. It can handle a wide range of graphs, including many that other solvers find particularly hard to solve.

Another example [46] proposes a portfolio of CPU and GPU algorithms. The approach drastically limits memory bandwidth constraints and avoids other typical portfolio fragilities as CPU and GPU versions often show a complementary efficiency and run on separated platforms.

4

MCS ON A SET OF GRAPHS

This chapter discusses existing approaches that tackle the MCS problems on more than two graphs, then introduces and describes a novel algorithm, *Alike*.

4.1 EXISTING APPROACHES

There is limited literature on the MCS problem applied to a set of graphs, as described by Hariharan et al. in 2011 [23], it is mainly used in molecular science, and many of the currently used methods are based on choosing a substructure that is present in one molecule or that is shared by two, and then confirming that it is present in the other molecules. More recently, Cardone [33] proposed to extend the McSplit algorithm to solve the problem of the maximum common subgraph on a set of graphs. He developed an exact as well as an approximate version consisting of the repeated application of McSplit first on a pair of graphs and then, once the result is obtained, between the current graph and the next one.

The following section proposes a new method to solve the MCS problem that is not based on the McSplit algorithm, but that reuses some of its components.

4.2 ALIKE

Alike is an algorithm for evaluating the similarity of a set of graphs and, as a byproduct, determining the degree to which each graph is similar to the entire set. The algorithm is approximated, as it compromises optimality for scalability. The approach builds a solution incrementally using a branch-and-bound procedure analogous to best-first search (BFS) [15]. The priority of a node in the search tree is determined by the mix of how good the current solution is and how much it may be further improved; the latter is also regularly updated during exploration by back-propagating the values of the new nodes.

Alike reuses the McSplit bound formula and bidomain data structure and enhances it by extending their applicability to graph

sets rather than pairs. The bound formula is used to inform its best-first strategy and to prune the search space.

It supports two or more input graphs, which can be undirected, directed, vertex- and edge-labeled. It can solve both the maximum common subgraph and maximum common connected subgraph problems. Given a set of graphs $\mathcal{G}$, *Alike* is able to evaluate its similarity $\Xi(\mathcal{G})$ (using equation 1 8) and find a reasonably large equivalence Q.

Similar to traditional BFS, *Alike* explicitly memorizes the solution tree and grows a new node at each step. However, it uses heuristic evaluations to prune away huge parts of the search space, closing off unexplored nodes; such a reduction may impede the search for the global optimum, but allows to solve more complex problems. Nonetheless, it can be easily modified to be exact.

Alike is designed with parallelism in mind and can be easily modified to perform as an exact solver or, conversely, to be highly resource efficient by employing a greedy and non-exhaustive strategy.

To be more specific, it uses best-first search to iteratively explore the space of possibilities Q_i and develop a solution Q. Best-first search is distinct from BFS (Breadth-First Search) and DFS (Depth-First Search) in that it utilizes problem-specific information to determine which search tree node to expand next. Best-first search is an informed search, whereas BFS and DFS are not. In this particular example, BFS investigates the tree of all possible solutions by maintaining two lists of nodes: the open nodes, which have been generated without but not yet fully branched, and the closed nodes, which have been fully expanded. In each phase, the best node on the open list is expanded, closed, and its descendants are added to the open list. Upon completion of exploration, the best solution(s) can be recovered by traversing the search tree.

4.2.1 *Architecture*

Alike tackles the problem with a very different approach from McSplit. While McSplit is a recursive algorithm, *Alike* explicitly stores the search tree in memory. This enables the best-first search strategy, that consists in expanding the most promising node at each step. In this section, the term *node* is used to refer to the search tree element, not the vertices of the input graphs. The most promising node at a given moment is defined as the node that

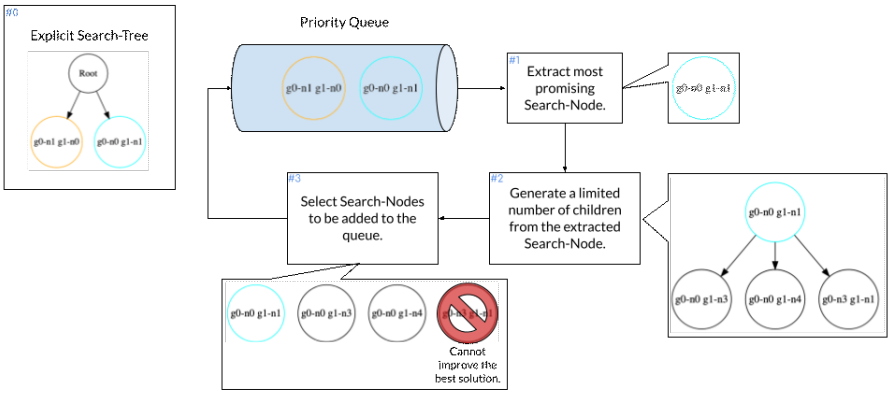


Figure 16: Diagram of the solution expansion procedure.

has the largest bound (calculated using 5 adapted to multiple graphs).

The pseudo-code for the entire procedure is reported in algorithm 2. The main procedure of the method consists in selecting the most promising search node candidate and executing a branch operation on it. This generates other candidates that are added to the priority queue. Finally, *Alike* discards some nodes, removing them from all future searches.

Figure 16 illustrates an high level view of the proposed search procedure.

4.2.2 *Vertex partitioning and upper bound computation*

As mentioned in section 3.4.1, McSplit uses a memory efficient data structure called *bidomain* to store the set of nodes belonging to the same adjacency class when considering the two input graphs. *Alike* uses a modified version of the data structure that supports multiple graphs. Partitioning the vertices by adjacency classes is critical to compute the upper bound that is used to drive the search.

Therefore, the upper bound of the size of the mapping M is the sum of the smallest number of occurrences for each label that occurs at least once for every $G_i \in \mathcal{G}$. The following upper bound formula is a generalization of equation 5:

$$\text{bound} = |M| + \sum_{l \in L} \min(|L_{G_1}|, |L_{G_2}|, ..., |L_{G_n}|) \qquad (6)$$

where L represents the collection of all labels that identify adjacency classes present in all graphs and

$$L_{G_i} = \{v \in V_{G_i} : \text{label}(v) = l\}.$$

Algorithm 2 *Alike*($\mathcal{G}$): Creates an explicit (tree-shaped) solution space and explores it, giving priority to promising nodes.

Input : A set of graphs $\mathcal{G}$.
Output: $solution_{best}$

29 $root\_node \leftarrow CreateRoot(graphs)$; `// Initialize the root of the tree with the the data structures needed (node partitioning, etc.).`
30 $candidates \leftarrow \{root\_node\}$; `// Initialize the queue with the root node.`
31 $solution_{best} \leftarrow 0$; `// Tracks best solution quality found so far.`
32 **while** $candidates \neq \emptyset$ **do**
33 $\quad candidates \leftarrow candidates\ NonImproving(candidates)$; `// Remove candidates that cannot generate children that improve the solution, i.e. they have a quality upper-bound that is less than or equal than the current best solution.`
34 $\quad candidate_{cur} \leftarrow SelectCandidate(candidates)$; `// Select the most promising candidate.`
35 $\quad new\_nodes \leftarrow Branch(candidate_{cur})$; `// Generate new children from the selected candidate.`
36 $\quad candidates = candidates \cup new\_nodes$; `// Add newly generated nodes to the candidate queue.`
37 $\quad solution_{best} \leftarrow max\{$
$\qquad Quality(solution_{best}),$
$\qquad Quality(new\_nodes[i]) : i = 1, ..., |new\_nodes|\}$
38 **return** $solution_{best}$

The vertex partition $\mathcal{P}$ is the extension of McSplit Bidomain data structure and is a set of sets of sets $\mathcal{P} = \{P_1, P_2, \dots, P_m\}$. Each set of sets P_j contains vertices from the different graphs $P_j = \left\{C_j^1, C_j^2, \dots, C_j^n\right\}$ with $C_j^g \subseteq V^g$; the vertices of graph G^g are partitioned into the different adjacency classes $C_j^g \subseteq V^g$. A solution Q_s is associated with the partition $\mathcal{P}_s$ that lists the vertices that could still be considered equivalent without violating any constraint. Thus, if $\mathcal{P}_s = \emptyset$, the solution Q_s cannot be extended by adding any equivalence set and the node v_s is a leaf in the solution tree.

4.2.3 *Search Nodes*

A *search node* represents a piece of the solution to the MCS problem. Each node v_i in the solution tree encodes an equivalence set

Q_i, except the root node v_0 that encodes an empty set $\emptyset$. The path from the root node v_0 to a generic node v_s defines an equivalence $Q_s = \{Q_{s_0}, Q_{s_1}, Q_{s_2}, \dots, Q_{s_n}\}$. The equivalence Q_s can only be extended with a new equivalence set $Q_{s_{n+1}}$ that does not make equation 4 invalid; as a result, only nodes encoding valid equivalence sets may be successors of v_i. As $Q_{s_0} = \emptyset$, it is safe to write $Q_s = \{Q_{s_1}, Q_{s_2}, \dots, Q_{s_n}\}$.

4.2.4 *Search Node selection*

At each iteration the most promising node in the priority queue of open nodes is selected and branched. A node is considered promising if it has the highest combination of solution quality and upper bound.

4.2.5 *Search Node Branching*

Once a node v has been selected, *Alike* generates a set of successors. This is done by selecting an adjacency class P_s from the vertex partition and then generating a fixed number of equivalence sets that haven't yet been produced by v. This can be achieved simply and efficiently by constraining the generation order of equivalence sets to be lexicographical.

If v is not explored exhaustively and is still promising after the first branching operation, it will be selected again until it either won't be promising anymore or it will have generated its possible successors.

4.2.6 *Pruning*

A search node v_i can be closed (or pruned) for two reasons:

- Their vertex partition is empty and thus cannot generate any new nodes.
- Their upper bound is lower than the best quality solution found so far.

4.2.7 *Parallelism*

Having an explicit search tree makes implementing a parallel version of *Alike* trivial. As reported in algorithm 3, the only high-

level modification needed is restructuring the search procedure such that it can run on multiple processes.

Algorithm 3 *Alike*($\mathcal{G}$): Creates an explicit (tree-shaped) solution space and explores in parallel, giving priority to promising nodes.

Input : A set of graphs $\mathcal{G}$.
Output: $solution_{best}$

39 $root\_node \leftarrow CreateRoot(graphs)$; `// Initialize the root of the tree with the the data structures needed (node partitioning, etc.).`
40 $candidates \leftarrow \{root\_node\}$
$solution_{best} \leftarrow 0$; `// Tracks best solution quality found so far.`
41 **while** $candidates \neq \emptyset$ **do**
42 $candidates \leftarrow candidates\ NonImproving(candidates)$; `// Remove candidates that cannot generate children that improve the solution, i.e. they have a quality upper-bound that is less than or equal than the current best solution.`
43 $candidates_{best} \leftarrow SelectTopKCandidates(candidates)$; `// Select the K most promising candidate.`
44 **for** $candidate_{cur} \in candidates_{best}$ **do in parallel**
45 $new\_nodes \leftarrow Branch(candidate_{cur})$; `// Generate new children from the selected candidate.`
46 $candidates = candidates \cup new\_nodes$; `// Add newly generated nodes to the candidate queue.`
47 $solution_{best} \leftarrow max\{$
$Quality(solution_{best}),$
$Quality(new\_nodes[i]) : i = 1, ..., |new\_nodes|\}$
48 **return** $solution_{best}$

4.2.8 *Heterogeneous Graphs*

A peculiar effect of allowing multiple best solutions to co-exist in the search tree is the "clustering" behavior which could be useful to partition heterogeneous sets of graphs. Figure 17 illustrates this behavior.

4.2.9 *Experimental results*

A prototype of *Alike* has been implemented in C++ [9] and it is distributed under *Apache License 2.0*. The programming language decision was influenced by the necessity to produce code

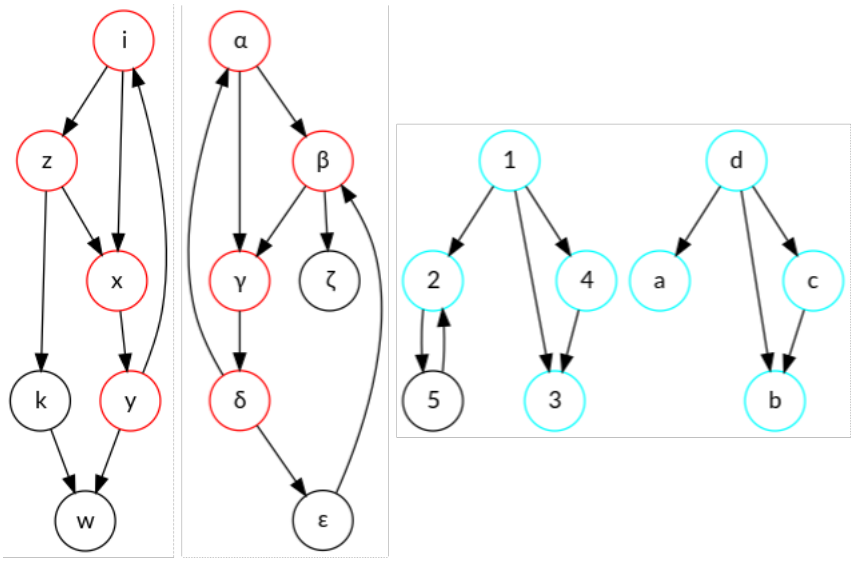


Figure 17: An heterogeneous set of graphs with two different common subgraphs of the same size highlighted.

with competitive performance, the availability of both low-level and medium-level implementation options for parallelism, the author's familiarity with the language, and access to precise performance metrics. The implementation has been compared with the original sequential and parallel McSplit implementation, and with a modified version of McSplit that can handle sets of graphs.

The tests have been executed on graph instances from the ARG database [2]. It consists of numerous classes of graphs that were randomly generated using one of six distinct generation algorithms and various parameters. A random sample of 40 graphs was selected from a subset of the dataset, consisting of pairs of graphs already prepared to have maximum common subgraphs of categories, i.e., pairs of graphs with maximum common subgraphs comprising 10%, 30%, 50%, 70%, and 90% of the original graphs' size.

The tests have been performed on a mid-2014 Apple MacBook Pro running MacOS Big Sur, with a 2.5GHz quad-core Intel Core i7 processor (Turbo Boost up to 3.7GHz) with 6MB shared L3 cache and 16GB of 1600MHz DDR3L onboard memory. All the solvers run with a 100 seconds timeout.

The variant of the problem tested is for the maximum common induced subgraph with undirected graphs, unlabelled nodes and edges. This is because these variants enforce constraints that restrict the search space, thus making the problem easier to resolve.

The results can be observed in Table ??, ?? and ??. *PMcSplit* is the original parallel McSplit algorithm, while *SMcSplit* is the sequential implementation. *PAlike* and *SAlike* are the parallel and sequential versions of *Alike* , respectively. *PAlikeLax* is a parallel version of *Alike* that doesn't constraint the elements of the solution to have a cardinality that corresponds with the number of input graphs. *MMcSplit* refers to the modified version of McSplit that can handle sets of graphs. *Nodes* indicates the number of branching operations applied in the algorithm.

Regrettably, the experiments [1] show that *Alike* is not able to keep up with the performances of McSplit. Since it hits the timeout most of the times without being able to find a better solution than McSplit. An important detail to note is the number of search nodes produced during the search: *Alike* generates consistently fewer nodes with respect to its counterparts. This is expected since, as mentioned in Section 3.1.7, algorithms with complex strategies usually move slower but, in turn, are more effective. However, it is promising to see that the CPU time in the parallel version of Alike is almost 8 times as much the Wall clock time. Suggesting excellent exploitation of the processor parallelism.

Table 1: Average comparison with McSplit versions on two graphs.

Graph Size	22.375	-	-	-
Algorithm	PMcSplit	SMcSplit	PAlike	SAlike
|Solution|	15.525	15.525	9.675	9.5
Solutions	1	1	4.2	4.025
Nodes	8750837.825	8906843	1447674.75	3227078.8
Nodes/|Solution|	563661.0515	573709.694	149630.4651	339692.5053
Time (s)	0.9395504	2.520845125	100.1419079	99.37229138
CPU time (s)	4.231347075	2.519320125	100.0625433	757.5118585

Table 2: Average comparison with the modified McSplit version on 3 graphs.

Graph Size	22.375	-	-	-
Algorithm	MMcSplit	PAlike	PAlikeLax	SAlike
|Solution|	14.15	10.925	10.35	11.15
Solutions	1	1.425	3.35	1.4
Nodes	336891394.4	503798.9	835,417.70	290421.625
Nodes/|Solution|	23808579.11	46114.31579	80716.68599	26046.78251
Time (s)	51.59383	100.1017574	100.1	100.0336755
CPU time (s)	51.59383	779.1232493	776.72	99.96234168

Table 3: Average comparison with the modified McSplit version on 4 graphs.

Graph Size	22.375	-	-	-
Algorithm	MMcSplit	SAlike	PAlike	PAlikeLax
|Solution|	12.4	10.075	9.4	9.825
Solutions	1	1.95	2.825	2.45
Nodes	686781490.3	61905.05	133642.55	363779.275
Nodes/|Solution|	55385604.06	6144.421836	14217.29255	37025.88041
Time (s)	86.74025668	100.0129063	100.0225639	100.0459582
CPU time (s)	86.74025668	99.94120058	779.3438776	781.3665515

5 GNN-BASED HEURISTICS

5.1 GRAPH NEURAL NETWORKS

This section describes the machine learning models that were utilized to enhance McSplit. First, one of the simplest machine learning models, the perceptron, is discussed. The fundamental principles of machine learning are also reviewed. The multilayer perceptron, the simplest neural network architecture, will then be described (Section 5.1.1). Lastly, Section 5.1.2 focuses on graph-specific machine learning techniques, with an emphasis on Graph Neural Networks (Section 5.1.3).

5.1.1 *Multilayer Perceptron and Machine Learning Fundamentals*

The perceptron is a mathematical algorithm inspired by biological neurons. Real neurons receive constant signals, and each signal activates a subset of the neuron's synapses. The activated synapses determine the significance of the input signal. The output is then transported along an axon. In a perceptron, the input is simply a vector of numbers, and the synapses are represented by another set of numbers known as *weights*. A perceptron is a binary classifier, meaning that, given an input, it can determine whether or not it belongs to a particular class.

Typically, a perceptron's input is a data point where each feature is represented by a number. For instance, analyzing images with a 256×256 resolution, a single image could be represented by a 256×256 vector where each number represents the pixel's color. Then, each element of the input vector is multiplied by its weight. The output is obtained by summing all the results with a bias and feeding them to a nonlinear activation function.

The perceptron is a machine learning model whose purpose is to learn the proper weights. Suppose each input either belongs to a class or does not; two possible labels exist:

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

After processing an input instance, the perceptron will *predict* its label. The objective of the learning process is a set of weights

that minimizes the number of errors on a set of general input instances.
Before being deployed, a perceptron undergoes three phases:

- A training phase, where weights are adjusted.
- A validation phase, where *hyperparameters* are fine-tuned.
- A test phase: where the performance is evaluated.

To execute the three phases, a dataset, or collection of input samples, is required. Each step will utilize a distinct subset of the initial dataset. Typically, the majority of the dataset is used during the training phase.
The training phase consists of feeding the training dataset to the perceptron, which was initialized with random values. A prediction will be obtained for each training set input instance. It's also assumed that the correct prediction for each instance is already known. Given this, a *loss function* can be used to evaluate the performance. The loss function quantifies the difference between the predicted and expected labels. The training phase's objective is to minimize the loss function score. Another algorithm called *gradient descent* accomplishes this. At each step, the entire dataset is fed to the perceptron, and after computing the loss, the weights are updated in the direction of the steepest descent. The direction of descent is determined by computing the function gradient. This procedure is typically repeated until the loss function reaches a local minimum. When updating the weights, the step size must be set, also known as the *learning rate*. The learning rate is simply a number that multiplies the gradient value; it is also a hyperparameter of the problem and must be selected before the training phase begins. The validation phase is required to determine the optimal learning rate and optimal set of hyperparameters. The validation phase consists of selecting a predetermined range of acceptable values for each hyperparameter, and training the perceptron with all possible combinations of those values. Each trained perceptron is evaluated on a subset of the dataset referred to as the validation set. The best perceptron is retained and used for testing. In the testing phase, the perceptron is evaluated on an additional portion of the dataset. It is crucial that the model has never encountered these data; otherwise, the results would be skewed. Figure 18 provides a graphical representation of a perceptron.

Multiple perceptrons can be stacked: the resulting architecture is the simplest type of neural network, known as *multilayer*

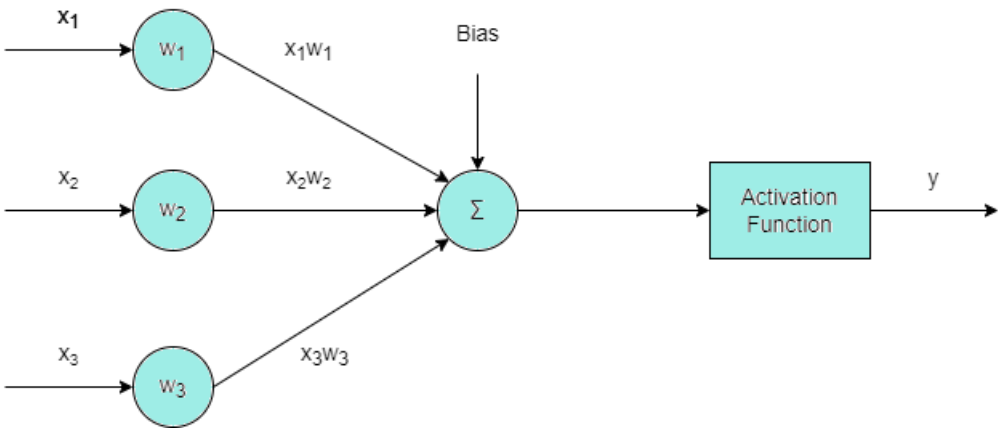


Figure 18: The scheme of a perceptron.

perceptron (MLP) [24]. Every perceptron constitutes a layer. Except for the first layer, which works directly on the input, each subsequent layer uses the output of the previous one. Each layer has an output, and the output of the final layer is the final result. A minimum of three layers is required for a deep neural network, which is one layer more than the input and output layers. The layer in the middle is known as a hidden layer, and there can be an arbitrary number of them. Training a MLP is considerably more time-consuming than training a single perceptron. Since there is a nonlinear activation function following each perceptron, this model can also approximate nonlinear functions. A graphical representation of the multilayer perceptron is shown in figure 19.

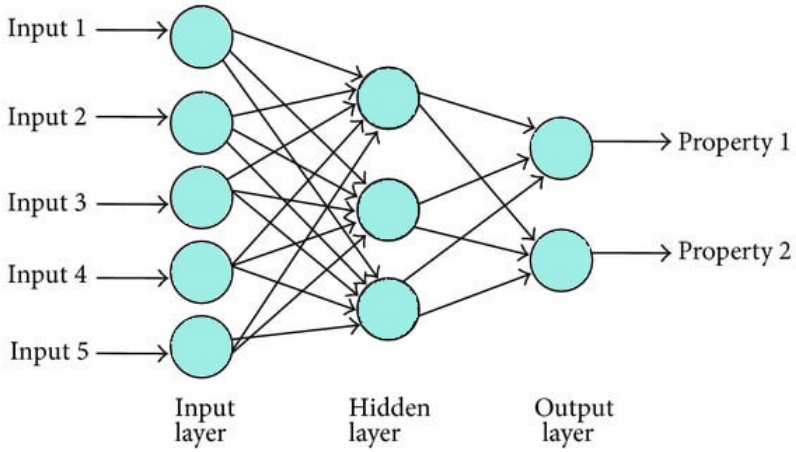


Figure 19: Illustration of a MLP.

5.1.2 *Machine Learning with Graphs*

This section describes the fundamentals of applying machine learning to graphs. The simplest method for approaching graphs is feature engineering, i.e., defining a feature vector "by hand". Methods capable of automatically learning a representation are examined later in this section. These techniques are known as representation learning. Specifically, the WL-kernel and the concept of *node embedding* will be introduced, which are closely associated with graph neural networks.

Feature Engineering Methods

The traditional machine learning pipeline is centered on feature design. There are two types of features: node level features (such as proteins with different properties) and structural features, which describe how these nodes are positioned relative to the remainder of the graph. By developing the appropriate characteristics, more precise predictions can be made. Therefore, the initial step involves transforming each graph into a vector of features. These vectors can be used to train any conventional classifier, including *support vector machines* and *random forests*. Some common node level features are:

- The degree of the nodes.
- Eigenvector centrality: the significance of a node is computed as the normalized sum of its neighbors' significance.
- Betweenness centrality: a node is considered significant if it is located on numerous shortest paths between other pairs of nodes.
- Closeness centrality: a node has closeness centrality if it has a short shortest path length to all other nodes.
- Clustering coefficient: indicates the connectivity of a node's neighbors.
- Graphlets: count pre-specified graphs in a node's neighborhood.

Graphlets are specifically rooted, connected, and non-isomorphic graphs. A graphical representation is provided in figure 20. A graphlet degree vector can be constructed to describe a node using graphlets by selecting a node and counting every possible

graphlet it participates in. Each vector cell will contain the final count for a particular graphlet. Graphlets exemplify structural characteristics, whereas node centrality measures more closely resemble node characteristics.

On occasion, characteristics that define the structure of the entire graph may be needed to calculate the degree of similarity between two given graphs. The concept of similarity can vary. Typically, *kernel methods* are applied to solve this issue.

A kernel is a function that measures the degree of similarity between two things (graphs); it accepts two vectors as input and returns a value between 0 and 1. Specifically, the kernel function represents the dot product between a specific representation of a feature from each of the two graphs. The Weisfeiler-Lehman kernel (WL), closely related to graph neural networks, will be discussed.

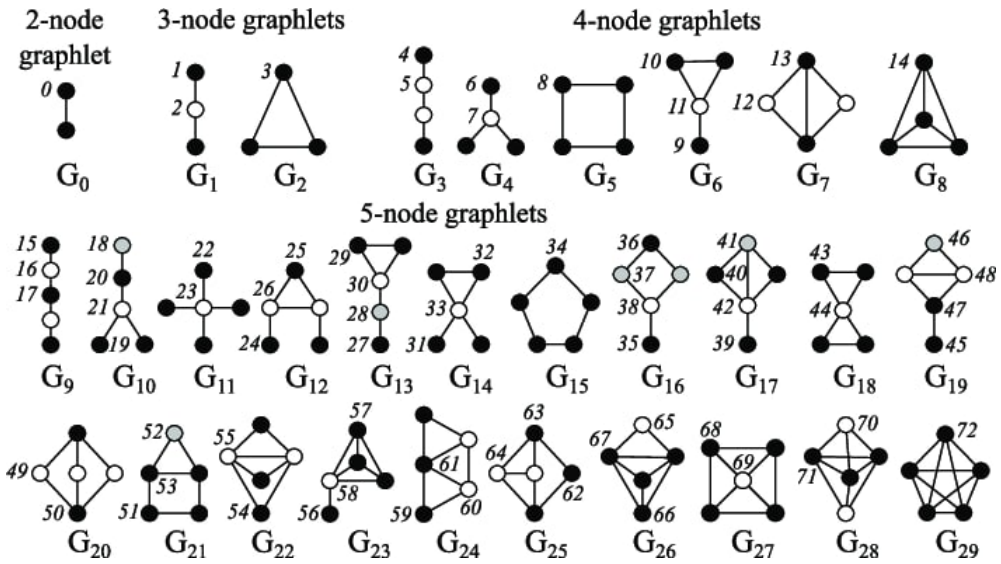


Figure 20: All possible graphlets with up to five nodes.

WL-kernel

The concept of *bag of node degrees* is generalized by the WL-kernel [58]. A bag of node degrees is a vector that counts the number of nodes with a certain node degree value within a graph. An example of a bag of node degrees is $V = [2, 0, 4]$ signifies that the depicted graph contains two nodes with degree 1, zero nodes with degree 2, and four nodes with degree 3. WL-kernel aims to iteratively expand the node's vocabulary by generalizing the node's degree by gathering degree information from a k-hop neighborhood.

The algorithm that accomplishes this is known as *color refinement*.

Given a graph $G = (V_G, E_G)$, initially a color $c^{(0)}(v)$ is assigned to each node $v \in V_G$. At iteration k, the node colors are modified using the following formula:

$$c^{(k+1)}(v) = HASH\left(\left\{c^{(k)}(v), \left\{c^{k}(u)_{u \in N(v)}\right\}\right\}\right) \tag{7}$$

The HASH function must be bijective, i.e., it must map distinct inputs to distinct colors. Consequently, at step k, the color $c^{(K)}(v)$ summarizes the structure of a node's k-hop neighborhood. After the final iteration, a bag of color vectors is computed. Each element of the vector represents the total number of nodes that possess a particular color. This process can be repeated for a second graph, and the dot product between the two bags of color vectors can be computed to derive a similarity measure. Figure 21 depicts a graphical example.

Representation Learning

With the help of representation learning techniques, it's possible to try and eliminate a challenging step in the machine learning pipeline: the necessity of feature engineering. Every node will be mapped to a d-dimensional feature vector through a learned function. The obtained vector is referred to as an *embedding*. Nodes that are comparable in the graph must also be close in the embedding space. Nodes that are dissimilar within the graph should also be dissimilar within the embedding space. Three things must be defined:

- A measure of similarity between nodes in a graph.
- An encoder: a mapping function between nodes and the embedding space.
- A decoder: a function that converts an embedded value to a similarity score (usually the dot product).

Encoder parameters must be optimized so that the embedding space similarity function is as close as possible to the graph similarity function. Typically, the dimension of an embedding vector ranges from 64 to 1000. Before discussing deep learning and graph neural networks, *node2vec*, a "shallow" embedding method, will be introduced.

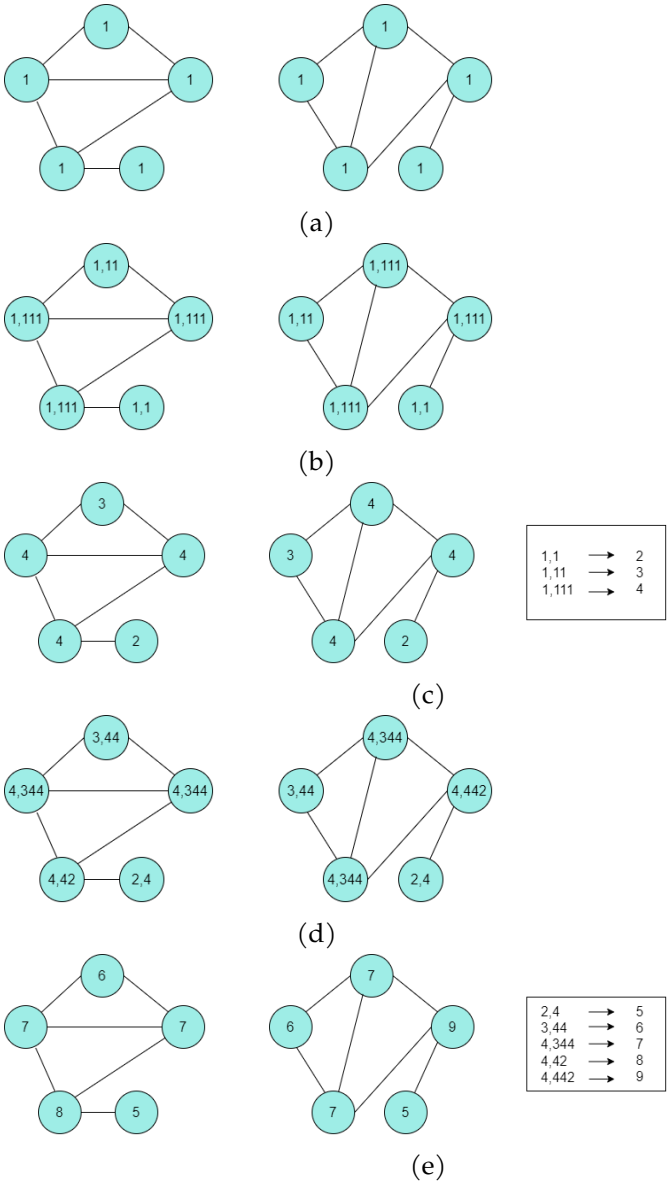


Figure 21: WL-Kernel application example

node2vec

The core element of node2vec [20] is to matrix computation. Each column of the matrix will correspond to a distinct node embedding. The node embeddings are optimized directly, while with GNNs, the *parameters* of the network are optimized, similar to the multilayer perceptron. The objective is to identify the optimal matrix, a set of embeddings whose dot product represents the chosen concept of similarity. node2vec computes embeddings using the *random walk* algorithm. A random walk is a walk that has been generated using a random strategy. For example, an im-

partial random walk may consider all neighboring nodes equally likely to be the next step.

The main random walk optimization steps for embedding space are summarized below:

- Perform a predetermined number of random walks from each node.
- For each node, the neighborhood is compiled. In this case, the neighborhood is the multiset of nodes visited by random walks.
- Given a node, optimize the embedding space to predict its neighborhood.

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

$$\sum_{v \in V_G} \left(\sum_{u \in N_R(v)} \left(-log \left(P \left(u | z_v \right) \right) \right) \right) \tag{8}$$

Where $N_R(v)$ represents the neighborhood of node v as determined by the random walk method R, and z_v is the embedding vector of node v. A softmax function may be used to represent the probability $P(v|z_u)$:

$$P \left(u | z_v \right) = \frac{exp \left(z_v^T z_u \right)}{\sum_{n \in V_G} \left(exp \left(z_v^T z_n \right) \right)} \tag{9}$$

The softmax function merely converts a vector of real values to probabilities by normalizing them so that they add up to 1. The gradient descent approach can then solve this optimization problem. Since the number of random walks in this scenario might be very large, gradient descent is typically computed on batches of training samples. The name of this algorithm is *stochastic gradient descent*. Now a random walk strategy must be defined. *Deep walk* [44] is an embedding technique that employs random walks without bias. This technique is generalized by node2vec by adding two hyperparameters. Adding the in-out variable q biases the random walks toward traversing the graph in depth (as in a depth-first search) or by breadth (as with a breadth-first search). A return parameter p is introduced to adjust the likelihood that the random walk will return to a previously visited node.

5.1.3 *GNN*

The multilayer perceptron is a reasonably generic framework for deep learning. It is compatible with practically any data structure that can be represented as a vector. In deep learning, special-purpose architectures that utilize a data structure's specific properties may be required in order to create significantly more accurate predictions with less training time. Convolutional neural networks (CNNs) are an example of such deep neural networks. CNNs are optimized for image-based applications. While images can be represented as vectors, spatial information included within the image itself is not optimally utilized with this representation. Moreover, images typically exhibit repeated patterns (e.g., all the windows on a building are identical), therefore it is advantageous to share neuron weights. The weight distribution between neurons is accomplished by multiplying the input by a sliding weight matrix. Another example is recurrent neural networks (RNNs), optimized for sequences. Text analysis is the most valuable use case. In the same way CNNs capture spatial information, RNNs capture time or ordering information (words are one after the other).

Graphs have a unique data structure, and their characteristics may be used to make more accurate predictions. Graphs are considerably more general: images can be seen as planar grid graphs (Figure 22a), and planar chain graphs can model sequences (Figure 22b). Therefore, GNNs are a generalization of both CNNs and RNNs.

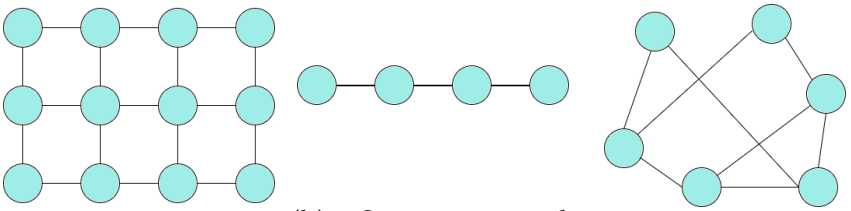
The preceding section discussed "shallow" techniques. These approaches have some drawbacks, including:

- Not a collection of parameters, but the embeddings themselves are optimized. By adjusting parameters, a type of "weight sharing" is obtained. The same set of settings can create two different node embeddings.

- Node features are not supported.

- They are transductive, meaning they can only construct node embeddings for nodes that were observed during training.

.



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

Figure 22: Graphs are more general than images or text.

Graph Convolution

The primary objective of graph convolution is generalizing the concept of convolution from CNNs [71]. A sliding window cannot be used because graphs lack a fixed notion of locality (e.g., a picture has a top-right corner). Each node will "send a message" to its neighbors through message passing, and each message is multiplied by a weight matrix W. Finally, the totals are tallied. Each node will have its own *computational graph* due to the necessity to specify the flow of messages. A computational graph is constructed by unfolding the target node's neighbors a predetermined number of times. Each unfolding specifies a GNN layer. Figure 23 depicts a graph on the left and the computational graph for node 1 on the right.

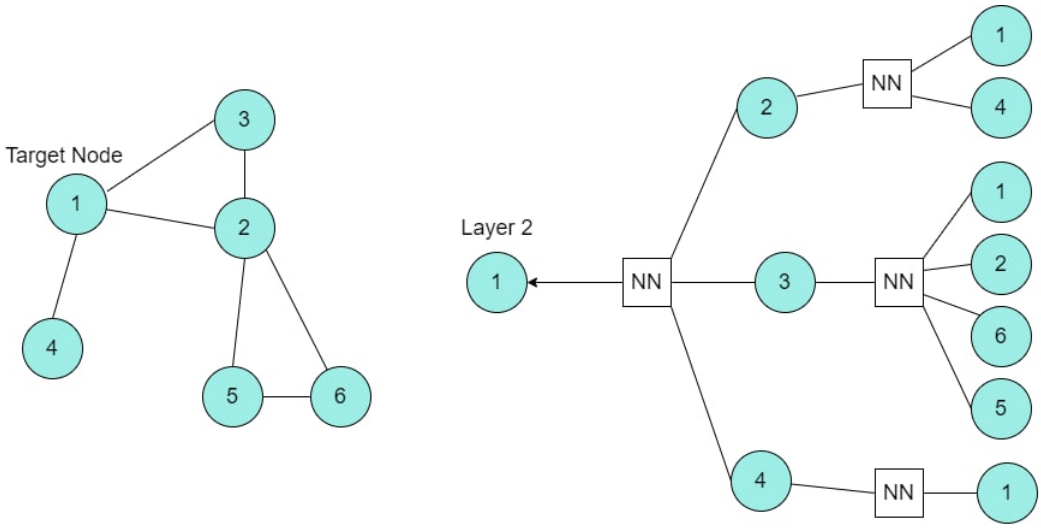


Figure 23: Computational graph.

Each node will have its own computational graph based on its vicinity. Each node will transmit messages to the next layer, after which they will be collected and then processed by a neural

network. The aggregation step is vital in GNNs, given that it can be executed in several ways. For instance, the average of all the messages could be used. Starting with $h_0 = x_v$, where h_i is the feature vector at layer i and x_v is the original feature vector input, the following equation is used to compute the feature vector (or embedding vector) at layer $i \neq 1$:

$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \left(\frac{h_u^l}{|N(v)|}\right) + B_l h_v^l\right), \forall l \in \{0, ..., L-1\} \tag{10}$$

The σ sign denotes the final stage activation function. W and B are the neighborhood aggregation weight matrix and a weight matrix used to change the target node embedding vector, respectively. Using any loss function and the gradient descent approach, it is possible to train both matrices, which are shared by all nodes. When neighbor aggregation is expressed in terms of matrix multiplication, it can be performed more efficiently:

$$H^{(l+1)} = \sigma\left(D^{-1} A\, H^{(l)}\, W_l^T + H^l B_l^T\right) \tag{11}$$

- H^l is the embedding matrix: it contains all node embeddings at a given layer l.
- A is the adjacency matrix.
- Each node's degree is represented by the diagonal matrix D. The inverse of D is employed so that the diagonal contains $1/degree$.
- W and B are the weight and bias trainable matrix.

This operation yields a product between matrices rather than a summative product.

Design choices in a GNN

When developing a graph neural network, there are various options available. Two operations comprise a GNN layer: message transformation and aggregation. Each distinct GNN architecture defines these two operations differently. Messages can also be changed before aggregation, for instance via matrix multiplication. The aggregation operator must be independent of permutation. The manner in which GNN layers are stacked is also a design option. Layers of a GNN can either be linked together or skip connections can be created. When layers are connected in a

chain, each layer l receives inputs from the layer $l-1$ corresponding to it. A skip link enables the reception of information from any previous tier. In conclusion, the computational graph outlined in the previous section can be extended in order to discover more accurate embeddings. The following will examine the most popular GNN layer types and design possibilities in detail.

Graph convolutional networks (GCNs) [71] were one of the first GNN layers to be proposed; it is denoted as:

$$h_v^l = \sigma\left(\sum_{u \in N(v)} W^l \frac{h_u^{l-1}}{|N(v)|}\right) \tag{12}$$

In this situation, the message is calculated using $m_u^l = W^l \frac{h_u^{l-1}}{|N(v)|}$. Specifically, at each layer l, the message m_u^l computed by node u at layer l is the product between the embedding vector of the previous layer h^{l-1} and a weight matrix W^l. The result is then normalized by the neighborhood size of that node. The aggregation operation is summation; the messages from all neighbors are added together. The GNN layer is defined as follows in *GraphSAGE* [21]:

$$h_v^l = \sigma\left(W^l \cdot CONCAT\left(h_v^{l-1},\ AGG\left(\left\{h_u^{l-1},\ \forall\ u\ \in N(v)\right\}\right)\right)\right) \tag{13}$$

In this instance, there is no single definition for the aggregation function. GraphSAGE permits various types of aggregation, including mean (taking a weighted average of all neighbors), pool (transforming the neighbor vector and then applying the mean or max function), and other advanced techniques that will not be covered (such as long short term memory). Additionally, message aggregation is performed in two phases. Inside the *AGG* function, only the neighbors of the node v are considered initially. The result is then concatenated with the embedding of node v.

Graph Attention Networks [65] are a powerful kind of GNN. In both GCN and GraphSAGE, all of the target node's neighbors are equally important. GATs are enhanced by the presence of a learnable attention matrix $a_v u$ that enables the network to determine which neighbors are more significant. Many traditional learning modules can be inserted between the layers of a GNN. The *batch normalization* and the *dropout* modules will be discussed below.

When training a machine learning model, input data is mapped to the appropriate output. Typically, it is assumed that the input

distribution remains constant throughout training. However, as each layer in neural networks is dependent on the previous layer's outputs, this assumption may not hold. By re-centering the node embeddings to zero mean and rescaling the variance to unit variance, *batch normalization* assists in stabilizing the networks. The batch normalization is accomplished in two steps: first, the mean and variance of the embeddings are computed; then, the embeddings are normalized using the computed mean and variance. However, normalization can diminish the expressive capacity of neural networks. In order to address this issue, two learnable parameters are added to the formula, allowing the network to select arbitrary values for the mean and variance.

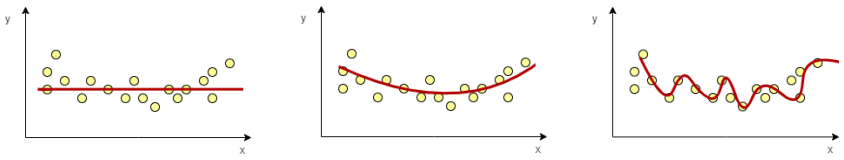
Dropout is a technique used to reduce the effect of *overfitting* in neural networks [61]. Every machine learning model trains and tunes its parameters using available data (the training dataset) and then applies the same rules to new data. These models are only useful if they can generalize. The generalization power may decrease when a model becomes increasingly adept at comprehending the training dataset.

This is because the training dataset frequently has a different distribution than the original data. The more information is available, the better the actual distribution can be approximated. However, most practical cases contain significantly less information than is required. If overtrained, the model will learn to match the training data precisely . This typically results in larger errors when evaluating the model with unknown data.

When computing the output of a neural network layer, the key idea is to eliminate a certain proportion of neurons. This prevents certain neurons from becoming more important during training. The method consists of three phases:

- Assigning a dropout rate. This will determine the proportion of neurons to be removed.
- Remove neurons at random until the desired number of neurons is reached.
- The output is calculated using only the remaining neurons.

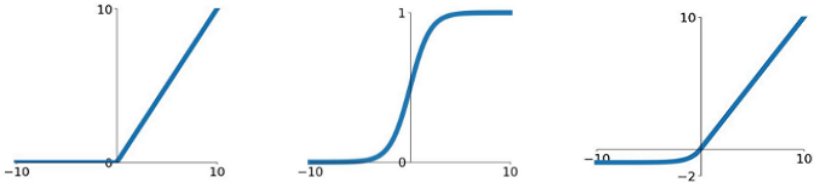
Thus, all neurons are allowed to contribute to the final output. In GNNs, dropout is applied during the phase of message transformation.



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

The most popular *activation functions* used in GNNs are:

- The *Rectified linear unit* (ReLU) is defined as $ReLU(x_i) = max(x_i, 0)$ and is the most commonly used.
- The *Sigmoid* is defined as $\sigma(x_i) = \frac{1}{1+e^{-x_i}}$ and is often employed when the range of the embeddings must be constrained.
- The *Parametric ReLU* adds a trainable parameter to the ReLU and is defined as $PReLU(x_i) = max(x_i, 0) + a_i min(x_i, 0)$. Typically, it outperforms standard ReLU.



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

Graph augmentation: as previously stated, each node defines a computational graph. A computational graph is a graph that illustrates how message passing occurs for a given target node. As discussed, a computational graph can be created by simply unrolling a node's neighborhood. Nonetheless, additional alternatives exist. The typical computational graph is unlikely to be the optimal method for computing embeddings. Several problems plague standard computational graphs. If the graph is insufficiently dense, message transmission will be inefficient. Adding virtual nodes or edges to the graph to enable more message passing solves this issue. Using virtual edges to connect 2-hop neighbors is a possible solution, and it is particularly useful when constructing bipartite graphs.

It is also a good idea to add a virtual node that connects to every other node in the graph so that the distance between each node is two. This makes the graph considerably less sparse and

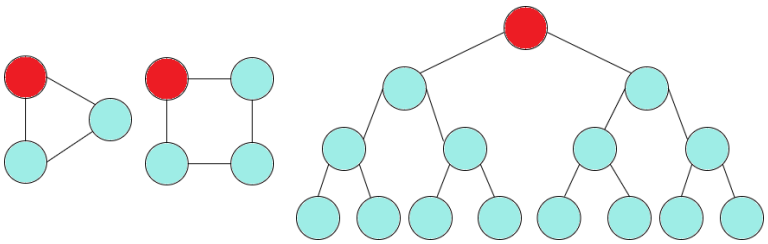


Figure 26: Red nodes in Figures 26a and 26b will both generate Figure 26c's computational graph.

is advantageous for message transmission. If the graph is too dense, it may be advantageous to disregard the neighborhood of every node. Typically, a different portion of a node's neighborhood is sampled at each layer. When each node has a large neighborhood, this is performed to reduce the computational cost of passing messages. Also, if a graph is too large, it may be challenging to fit it on GPUs. As the first message at layer 0 is composed of the node feature vector, node characteristics are also crucial when computing messages. If nodes are missing features, it may be beneficial to add them. A common approach is to assign each node a constant value, such as 1. This is particularly helpful in inductive settings (when generalization on unseen graphs is desired). The GNN will learn from the graph's structure despite all nodes being identical. For transductive settings, assigning an ID to each node and using it as a characteristic is possible. This procedure is more expressive because it stores node-specific information, but it does not generalize well because a GNN cannot determine the identifier of an unseen node. Without node features, it is much more difficult for a GNN to differentiate between nodes. For instance, the red nodes in figures 26a and 26b will produce the identical computational graph (Figure 26c).

The cycle count could be a possible augmented feature vector to solve this problem. Each node is aware of the length of the cycle in which it resides. In this manner, the red node in Figure 26a will have a value of 3, as its cycle has a length of 3 units. Figure 26b's red node will have a value of 4. Now, a GNN can distinguish between the two nodes because their characteristics are distinct (the structure of the computational graph will remain the same).

Various types of problems can be modeled using GNNs. Reducing a problem to recognizable patterns or *prediction tasks* is advantageous. The prediction task is typically stated in one of the following ways:

- At the node level, predict a discrete (classification) or continuous (regression) value for nodes.
- Classification or regression on node pairs or the prediction of new edges.
- Classification or regression across the entire graph.

The last layer of a GNN will generate embeddings. The resulting embeddings are used to make predictions for node-level tasks. Pairs of the original embeddings can be concatenated to create new embeddings for link-level tasks. The alternative is to use the dot product between two embeddings. In this manner, a simple binary value representing the presence or absence of an edge can be predicted. For tasks at the graph level, embeddings must be aggregated in alternative ways. Common techniques include calculating all embeddings' mean, maximum, or sum. A great deal of expressive power is lost by naively employing these techniques.

Suppose that graph G has a node embedding vector $z_g = [-1, -1, -2, 1, 3]$ and graph H has a node embedding vector $z_h = [-10, -10, -20, 10, 30]$, while the sum operation is used to aggregate these embeddings. Both G and H will have an embedding of 30 as a result. Consequently, the GNN will be incapable of differentiating between these two vastly distinct graphs. This issue can be solved by aggregating embeddings hierarchically. The result of aggregating the first two nodes of each graph is then fed into an activation function. The final three nodes undergo the same process, and the final two results are aggregated once more.

Setting up a *graph dataset* is trickier than others, e.g., an image dataset. In an image dataset, each image is an independent data point. The prediction of an image depends solely on the image's unique characteristics. When classifying nodes, the configuration varies. Each node is linked to its neighbors by one or more edges (Figure 27). In this instance, other nodes will impact the target node's prediction. Whether the task is *transductive* or *inductive* may affect how datasets are used.

In a transductive setting, only one input graph is available during all three phases (training, validation, and testing) of a neural network training pipeline: training, validation, and testing. However, graph nodes are divided into training, validation, and test groups. During training, embeddings are computed using the entire graph, but only the labels of the training nodes

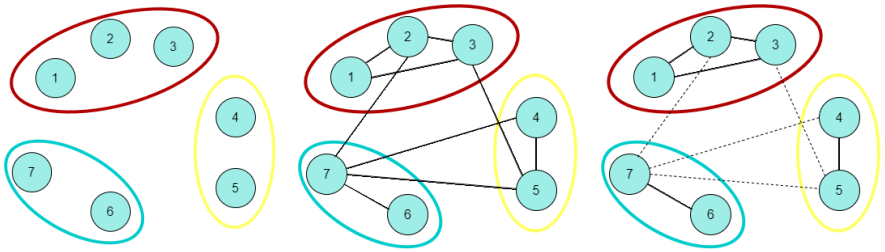


Figure 27: Seven data points were separated into training (red), validation (yellow), and test sets (light blue). Figure a depicts an image dataset in which each point is independent. Figure b depicts a graph dataset in which nodes are actually interconnected. Figure (c) depicts a graph dataset that was divided inductively. The initial graph is divided into three separate graphs.

are used. In the same way, only validation nodes are evaluated during validation. In an inductive setting, the dataset consists of multiple graphs. The objective of the inductive setting is not to predict missing node labels but rather to generalize to unobserved graphs. In this instance, the dataset is divided into training, validation, and test, similar to how an image dataset would be divided. There are node-level and link-level tasks for both inductive and transductive settings, but only the inductive setting has well-defined graph-level tasks.

Expression potential of GNNs

The expressive power of a GNN resides in its capacity to differentiate between various graph structures [68]. In general, a GNN cannot distinguish symmetric or isomorphic nodes in a graph. The embedding of each node is computed according to the message passing paths specified by its computational graph. If two nodes share the same computational graph structure, so will their embedding. This is even more problematic if nodes have identical characteristics. The aggregation steps introduce a new difficulty that could diminish the expressive power of the GNN. After aggregation, two distinct sets of messages can produce the same output. This causes various graph structures to be mistaken for the same one.

Mean pooling is the mean operator used in graph convolutional networks; it yields the same output if the proportion of each label's inputs is the same. Suppose that there are two labels, green and red, encoded by the vectors $(0, 1)$ and $(1, 0)$, respec-

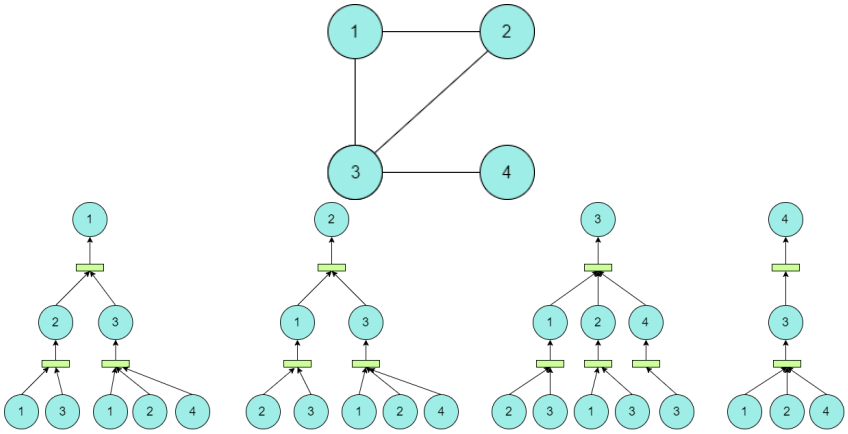


Figure 28: The two symmetric nodes 1 and 2 generates the same computational graph.

tively. If two green and two red messages are received before the aggregation step, the output obtained by applying the mean to each element is $(0.5, 0.5)$. The same output is produced in all circumstances where half of the messages are green and half are red.

5.2 GNNS FOR THE MCS PROBLEM

In this section, three different methods are proposed to enhance McSplit by using Graph Neural Networks. As previously mentioned, McSplit is an exact algorithm optimized to find the MCS between two input graphs in the shortest time possible. As the MCS problem is $\mathcal{NP}$-hard, this approach will not scale well to larger graphs. The aim is to optimize the algorithm so that it can find higher-quality solutions in a short amount of time. To accomplish this, Three distinct methods that utilize McSplit as their foundation are proposed to accomplish this. As stated previously, McSplit employs two heuristics:

- The bidomain heuristic, that gives precedence to the bidomain with the smallest
  $max(|G|, |H|)$
- The node ordering heuristic: higher degree nodes are tried first during the search.

The primary objective of these heuristics is to reduce the tree search size, allowing the branch and bound algorithm to perform more pruning. However, there are a few disadvantages. The node degree is an overly simplistic metric; consequently, most

heuristics are based on random factors, i.e., the alphabetical order of nodes. In addition, McSplit adheres to a *fail-first* paradigm, frequently utilized among branch and bound algorithms. Its goal is to reach leaf nodes in the search tree as quickly as possible so that a more aggressive pruning can be performed later. This is beneficial for McSplit's goals because it is interested in the exact solution.

Section 5.2.1 introduces NeuroMatch. Then, using the representational power of Graph Neural Networks, two "smarter" node ordering heuristics will be implemented (section 5.2.2). Subsequently, McSplit will be modified significantly by introducing a best-first method that selects node pairs directly (section 5.2.3). Finally, a custom neural network that learns a node's likelihood of being part of a large MCS solution will be presented in section 5.2.4.

5.2.1 *NeuroMatch*

NeuroMatch is a graph neural network (GNN) that computes and imposes an order constraint on graph embeddings, allowing for fast graph matching calculation [34].

Given a graph $G = (V, E)$, NeuroMatch learns embeddings by selecting one node $v \in V$ at a time 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.

As previously described, the k-hop neighborhood of a given node $v \in V$ is defined as the subgraph induced by the set of nodes that includes v and all nodes that can be reached from v through a path shorter than k. Graph embeddings are learned by enforcing an order constraint, as the geometry of the embeddings represents the relationship between subgraphs. This also enables matchings to be computed by merely comparing the components of two embeddings.

NeuroMatch satisfies the following four requirements for subgraph relationships:

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

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

- *Intersection set*: The intersection of the subgraph sets G and H contains all subgraphs shared by both sets.
- *Non-trivial intersection*: the intersection of any two graphs contains the trivial graph, that is, a graph with one node and no edges.

In practice, a graph G with D-dimensional embedding vector Z_G is considered 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.

$$\forall i \in [1, D] : Z_G[i] \leq Z_H[i] \iff G \subseteq H \tag{14}$$

To guarantee the previously indicated order limitation, NeuroMatch is trained with the maximum margin loss:

$$\begin{aligned} \mathcal{L}(Z_G, Z_H) = \textstyle\sum_{(Z_G, Z_H) \in P} E(Z_G, Z_H) + \\ \textstyle\sum_{(Z_G, Z_H) \in N} \max(0, \alpha - E(Z_G, Z_H)) \end{aligned} \tag{15}$$

where $E(Z_1, Z_2) = \max(0, |Z_1 - Z_2|_2^2)$. P and N are the positive and negative samples, respectively. Positive samples consist of graph pairs wherein the first graph is a subgraph of the second. The pairs that do not satisfy this requirement are referred to as negative samples. The loss is minimized when the subgraph relationship specified by E holds for pairings in P but is violated by at least *alpha* pairs in N.

The training technique is further improved by a curriculum learning scheme, in which the model is initially trained on easier instances that progressively get more difficult as the model loss stabilizes.

Figure 29 demonstrates that the outcome is an order embedding space. G is contained in H if it is located on the lower-left side of H in the order embedding space.

Embedding Computation

In this particular scenario, Embeddings for each node $v \in G$ are computed while taking their k-hop neighborhood into consideration. Sampling all neighbors up to k yields the related induced subgraph of G on which the embedding is calculated. The sampled subgraphs are sampled into batches before to loading them into the GPU memory, such that process is sped-up using the GPU parallelism.

This helps the GPU to spend less time in an idle state, as multiples of 32 threads form each thread warp, and having only one thread operating negates the GPU's benefits by leaving 31 threads inactive. NeuroMatch does not rely on a specific GNN and may utilize all the numerous cutting-edge models, including graph convolutional networks (GCNs), GraphSAGE, and graph isomorphism networks (GINs). In this work, an 8-layer GraphSAGE model is used.

Finding the optimal BFS depth

Given an anchor-node v, the parameter k allows choosing the depth of the breadth-first visit used to construct the k-hop neighborhood subgraphs. In other words, k represents the eccentricity of the sampled subgraph's anchor node. Similarly, a GNN with a predetermined number of layers j is employed. There is no relationship between j and k for the following reason: A GNN with j layers computes an embedding for a particular node by aggregating and manipulating information obtained from its j-hop neighbors.

Since initially a k-hop neighborhood subgraph is sampled using a BFS, if $k > j$, any nodes with a distance greater than $d > j$ from the anchor vertex will be disregarded. Therefore, the number of layers j must always be greater than or equal to k to ensure that information from all nodes propagates to the anchor.

Additionally, the value of k impacts the total performance. Indeed, even though the number of layers during the embedding computation is j, the value of k corresponds to the depth of the BFS technique that must be executed, and consequently, k increases the time cost of the procedure.

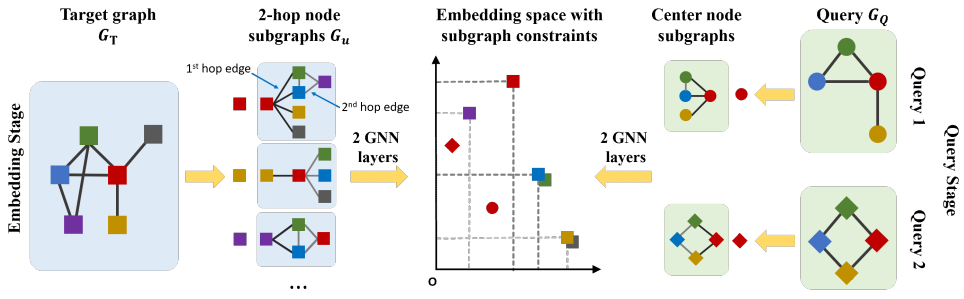


Figure 29: NeuroMatch architecture [69].

5.2.2 *GNN-based node ordering heuristics*

As previously discussed, McSplit first selects the bidomain with the least $max(|G|, |H|)$, i.e., by comparing the values produced by the formula for all potential bidomains. Then, the vertex with the highest degree is chosen within the selected bidomain. Given the graphs G and H, the method constructs the final $MCS(G, H)$ by matching one vertex of G with one vertex of H and sorting the nodes of G and H depending on their degree, so that high-degree nodes are matched before along the branch-and-bound process.

This strategy, despite its simplicity, has numerous downsides, primarily because the node degree does not capture graph structural information about neighbors, resulting in a large number of ties that are ultimately resolved using the irrelevant original lexicographic order of the nodes. On big graphs, this causes scalability concerns and inefficiency. This section presents an approach that utilizes the main concepts of McSplit and modifies its static heuristic by ranking nodes with 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), Consider a subgraph containing all k-hop neighboring nodes of v. Increasing or reducing the value of k provides finer or coarser information about G to the node embedding, respectively.

Norms represent concisely the order and size of each subgraph created through k-hop partitioning of the graph. Greater embedding norm values are connected with larger subgraph sizes. The cumulative cosine similarity is obtained by adding the cosine similarity between a node in the first graph and all other nodes in the second graph. This causes a node to be ranked higher when it has a large number of comparable nodes in the second graph, since it has a greater a priori likelihood of being part of a larger solution.

Using Algorithm 4 as a guide, the suggested heuristic substitutes the node degree heuristic in lines 9 and 11 of McSplit. Instead of the maximum degree, the priority method considers the maximum norm or cumulative cosine similarity. Similar to the original technique, the scores are generated during a setup phase before invoking the `mcs` function.

Algorithm 4 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*

```
49 if |M| > |incumbent| then
50     incumbent ← M
51 bound ← |M| + calc_bound(L);                          // Eq. 5
52 if bound ≤ |incumbent| then
53     return incumbent
54 bd ← select_bidomain(L);                    // bidomain selection
55 v ← select_left_node(bd);                       // node selection
56 remove v from bd
   for each w ∈ bd do
57     remove w from bd
       M.push((v, w));                          // Add new pair to M
58     new_L ← filter_labels(L, v, w);          // Generate new L set
59     incumbent ← mcs(G, H, M, incumbent, new_L)
       M.pop()
       add w to bd;                             // w is added back
60 if bd.V_{l,g} is empty then
61     remove bd from L
62 return mcs(G, H, M, incumbent, L)
```

Embedding norm

The first heuristic presented is based on the L2 norm. The Euclidean Norm is computed by evaluating NeuroMatch embeddings on the k-hop neighborhood of each node in both graphs. To be more specific:

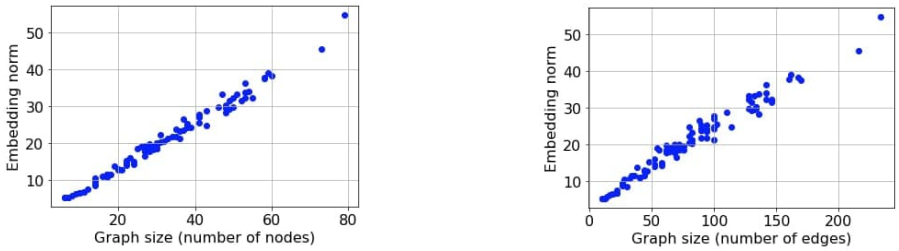
$$L2\,(Z) \quad = \quad \sqrt{\sum_{i=1}^{D} Z\,[i]^2} \tag{16}$$

where Z is a D-dimensional embedding vector. Due to the NeuroMatch order constraint, the L2 norm of these embeddings for nodes of large and dense graphs is greater. Instead of only considering the degree of a node, the size and order of the entire k-hop neighborhood are considered. Due to the NeuroMatch order constraint, the L2 norm of these embeddings for nodes of large and dense graphs is greater.

Experiment results are shown in figure 30 to demonstrate the validity of this hypothesis.

Using a BFS with a depth limit of 3, 100 subgraphs are sampled from a single graph with 100 nodes.

Then, for each graph, the mean value is plotted against the number of nodes, as shown in figure 30a, and edges (figure 30b).

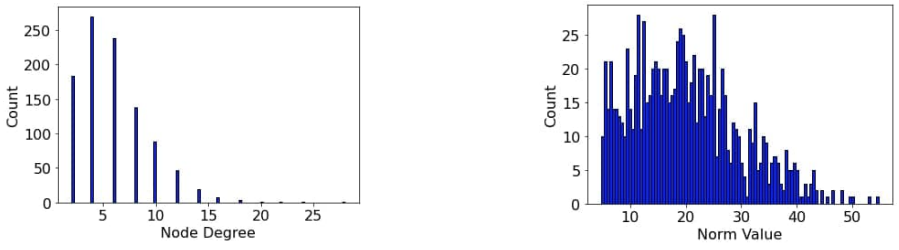


(a) (b)

Figure 30: The embedding norm increases with the number of nodes (a) and edges (b)

Figures 31a and 31b depict the degree and norm distributions computed for a set of 10 graphs with 100 nodes each. It is observable that the distribution of node degree is considerably less sparse. Since the graph is connected and contains 100 nodes, the degree values of the nodes could range from 1 to 100. In contrast, most nodes tend to have few neighbors, and the average degree is approximately 5. This is common in the graph domain, where nodes tend to cluster into small neighborhoods.

In addition, while the degree of a node is a discrete variable, the embedding norm is continuous. Since McSplit breaks ties lexicographically, the performance of the node degree heuristic is significantly more dependent on random factors.



(a) (b)

Figure 31: The node degree distribution (Fig. a) is much less sparse than norm value distribution (Fig. b)

Recomputing the Norm

As the objective is to give higher priority to nodes with a larger and denser neighborhood, it may be advantageous to periodically recompute the norms while the procedure is running. This allows the removal of already selected nodes from all neighborhoods. However, the computational cost overhead for recalculating the norm at each recursion step is so high that recalculating the norm consumes most of the budget time. Therefore, despite the fact that this strategy should maximize the efficiency of the selection heuristic and reach good solutions with fewer recursions, an algorithmic improvement is required to make it appealing. At every recursion call, McSplit chooses a new pair of nodes. Consequently, at each new call along the search tree, the newly selected pair of nodes is added to the previously selected set. When expanding the current solution and making a new selection, previously selected or unselected pairs are no longer considered. This consideration influences how and when the actual embeddings are computed (and possibly updated).

By only initially computing embeddings, the entire k-hop neighborhood surrounding each node is considered. However, as the search continues, it may be prudent to disregard nodes that are already part of the solution as members of any neighborhood. For example, the following situation is conceivable: A vertex v has a large neighborhood consisting primarily of nodes already included in the solution, whereas a vertex u has a small neighborhood consisting primarily of nodes available for future matchings. In this circumstance, the norm evaluated at the outset of the process may suggest an incorrect vertex selection by giving v precedence over u.

To avoid this issue, the most obvious solution is to (potentially) recompute embeddings after each pair selection and, concurrently, to ignore all nodes already selected in the current solution when sampling neighbors during the BFS. This solution corresponds to removing from the current solution all nodes from previously sampled subgraphs. Therefore, the nodes affected by this procedure will have lower standards and a lower priority as the selection procedure progresses. Moreover, this procedure will increase the cost of computing embeddings.

Experimentally, computing the embeddings only at the beginning of the process requires only 0.1 to 0.4% of the total budget time t. In contrast, recalculating the norms at each recursion

can consume over 95% of t, drastically reducing the number of recursions that can be performed by the algorithm.

A trade-off is necessary to balance time and precision in norm computation. One possible strategy is to recompute the norm if the algorithm fails to reduce the solution size within a predetermined amount of time or a predetermined number of iterations. This could be accomplished in various ways. The naive strategy would be to stop the algorithm at any branching point, recalculate the norm, and restart the algorithm with the new norms.

A more effective strategy could be to backtrack to a promising branching point before computing norms there. In [3], for instance, the algorithm backtracks to the branching point with the largest action space, i.e., the highest number of possible branches. This can also prevent the occurrence of a local optimum.

Cumulative Cosine Similarity

The objective is to compute a score for each node in both graphs, as in the previous section. The cosine similarity between two nodes represents their respective k-hop neighborhood similarity. Nodes with neighborhoods that are highly similar will have a higher similarity score. In this instance, the graph nodes are ordered according to the sum of the cosine similarity between a node of G and all nodes of H. The primary benefit of considering this metric is that the score of each node is determined by reasoning on both graphs. In contrast, the norm and degree-based heuristics only consider a single graph.

In figures 32 and 33, cosine similarity values for two pairs of graphs are displayed. The higher the cosine similarity, the more similar two graphs are.

In the first step, given two graphs G and H, the matrix M is derived, where M_{ij} is the cosine similarity score between nodes $i \in G$ and $j \in H$.

Then, the score of each node in G is determined by the sum of values on the corresponding row in M, whereas the scores of nodes in H are determined by adding over columns. Specifically, the following is computed for the nodes of G:

$$CSS\,(i) = \sum_{j=1}^{|H|} M_{ij} \; where \; M_{ij} = Z_i \cdot Z_j \tag{17}$$

Z_i and Z_j are embedding vectors of node $i \in G$ and node $j \in H$, respectively. This score indicates the frequency with which a subgraph sampled from one graph and represented by a node's k-hop neighborhood appears within the other graph. These nodes

have a greater likelihood of being a part of a large solution or being matched, so they have a higher priority.

(a) (b)

Figure 32: Two graphs with cosine similarity of 1.

(a) (b)

Figure 33: Two graphs with a 0.22 cosine similarity.

Neighborhood sampling and cosine similarity

The number of GNN layers has a significant effect on the informativeness of the cosine similarity. It can be demonstrated that the expressive power of a GNN does not increase monotonically with the number of layers, as is the case with other types of neural networks. This is a phenomenon known as over-smoothing [**chen2019measuring**], which also depends on the size of the graph.

Typically, larger graphs permit the addition of more layers before over-smoothing. Before embedding the graph, a k-hop neighborhood surrounding each node in the original graph is sampled.

This issue is also present in the proposed setting, as shown in figure 34. The figure's dataset contains graphs with up to 100 nodes. The analysis led to the conclusion that a k value of 1 or 2 may be optimal for this dataset.

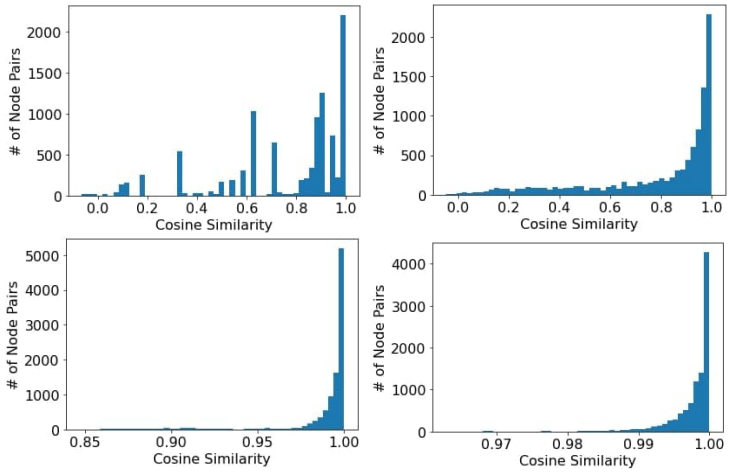


Figure 34: Distribution of cosine similarity scores computed for all node pairs in a 100-node sample graph taking into account its k-hop neighborhood, with k equal to 1, 2, 4, and 6 (Clockwise from top left). For k equal to 4, nearly all pairs have a similarity greater than 0.90, whereas for k equal to 6, nearly all pairs have a similarity greater than 0.99.

5.2.3 *Best-pair McSplit Implementation*

This section proposes a radical modification of the original McSplit algorithm, which uses only one heuristic instead of two. Again, the employed heuristic is based on cosine similarity. While the McSplit heuristic operates at the node level, the node selection is replaced by a node-pair heuristic, which scores node pairs directly. In McSplit, a node from the first graph is paired with each node from the second.

The nodes of both graphs are prioritized following the same heuristic. McSplit also subdivides nodes into different classes, representing a node connectivity pattern with respect to already selected node pairs. These classes are used to prune the search space further. 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, a different priority is given to node pairs. Cosine similarity is the score given to each pair, computed from the embeddings of both nodes. Initially, an attempt is made to select a pair from a node class determined by the McSplit class heuristic. Subsequently, the algorithm is modified to select the best pair among all classes, thereby replacing the original McSplit heuristics with the proposed node-

pair heuristic. McSplit classes continue to be utilized to reduce the search space.

Choosing the Most Similar Pairs Within a Selected Bidomain

Node pairs with greater cosine similarity are prioritized without affecting the McSplit bidomain heuristic, thereby replacing only the node ordering heuristic. Some modifications to McSplit are required to make this possible. The original implementation of McSplit selects a node from the first input graph and attempts to match it with all nodes from the second input graph that are in the bidomain determined by the bidomain heuristic. In this circumstance, node pairs are chosen directly. After selecting the bidomain, all node pairs within that bidomain are tested, with high cosine similarity scores being prioritized. Cosine similarity scores can be pre-computed and stored in a priority queue. However, since the chosen bidomain is unknown, a subset of the queue must be selected at each recursion level, corresponding to all node pairs in the chosen bidomain. In addition, as with the norms heuristic, the node embeddings are recalculated with each recursion call. The new embeddings are still computed based on the k-hop neighborhoods of the nodes but exclude already chosen nodes.

Selecting the Bidomain Pairs With the Highest Similarity

In this instance, the McSplit bidomain heuristic is disregarded, and a node pair is chosen directly. Because each node pair belongs to a single bidomain, the chosen bidomain is automatically derived. Before the recursive phase starts, the cosine similarity between every pair of nodes is computed and stored in a priority queue. Again, node pairs with greater cosine similarity are given precedence. All recursion levels use the identical copy of the priority queue. To facilitate this, the queue's last selected pair index is tracked. In other words, if the last selected pair was at index i, the first pair considered in the subsequent recursive call is the pair at index $i + 1$.

This also eliminates the need to evaluate permutations of the same solution. Observe that as the algorithm continues, not all pairs remain valid. Only pairs belonging to the same bidomain at that particular recursion level can be chosen, so even if the first pair at index $i+1$ is considered, it does not necessarily mean it will be chosen. In such a case, the next pair in the queue is evaluated

until a valid pair is discovered. Algorithm 5 demonstrates the entire procedure.

Algorithm 5 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*

63 **if** $|M| > |incumbent|$ **then**
64 $incumbent \leftarrow M$
65 $bound \leftarrow |M| + calc\_bound(L)$ **if** $bound \leq incumbent$ **then**
66 **return** *incumbent*
67 **while** $idx < queue.size()$ **do**
68 $new\_idx, v, w, bd \leftarrow get\_pair(queue, idx, L)$; `// Get next valid pair, its index and its bidomain.`
69 remove v and w from bd
 $M.push((v, w))$; `// Add new pair to M.`
70 $new\_L \leftarrow filter\_labels(L, v, w)$; `// Generate new L set.`
71 $incumbent \leftarrow mcs(G, H, M, incumbent, new\_L, queue, new\_idx)$
 M.pop()
 add w and v to bd; `// w and v inserted again in the bidomain.`
72 $idx \leftarrow idx + 1$
73 **return** incumbent

The function `get_pair` searches the queue beginning with the index *idx* for a valid pair. To validate a pair, it is necessary to know which bidomain each node belongs to. This information may change at each level of recursion. To improve the efficiency of checks, a dictionary mapping each node to its bidomain is computed.

5.2.4 *Custom Neural Network*

All previous efforts have focused on learning a score on nodes or node pairs that can correctly prioritize nodes so that a larger solution can be found more quickly. All of these approaches, however, are based on **unsupervised** machine learning. Learned node embeddings from the structure of the graphs are then used

to compute norm or cosine similarity scores. This section will propose a **supervised** learning method for acquiring these scores.

Unsupervised learning is based on learning embeddings directly from unlabeled data, i.e., it learns from the data structure only. Supervised learning learns from labeled data. The goal is to learn scores for node pairs. These scores are thought to be used together with the McSplit best-pair variation proposed in section 5.2.3.

The idea is to utilize a training dataset consisting of MCS problem instances and the optimal solution (computed by the original McSplit). The network should then learn to compute embeddings such that the notion of similarity between two nodes can be represented as a similarity between embeddings vector, i.e., two nodes that have been optimally paired should be close to one another in the embedding space. If two nodes have not been paired, they should be separated in the embedding space.

The similarity measure is a number between 1 and 0 and is interpreted as the probability that McSplit will choose a pair of nodes as part of the optimal solution. By prioritizing these pairs, if correctly learned, the best-pair version of McSplit should converge to the optimal solution more quickly.

The network, shown in figure 35, consists of two MLPs. Both graphs are inputs and separately fed to the first MLP. The MLP outputs are then combined and fed to a second MLP. The output is a probability matrix.

Using the MLP model, which requires a fixed-size input and is not optimized for graphs, is the major limitation of this method. GNNs could be utilized in the future to overcome this limitation. Figure 35 depicts a graphical representation of the proposed architecture. The Python implementation is illustrated in algorithm 6. The framework for deep learning is PyTorch. The `dim` value indicates the size of the graph input. If graphs with 10 nodes are analyzed, for instance, `dim` will equal 10, while the adjacency matrix will have dimensions of 10×10.

The output likelihood matrix could be used in conjunction with the best-pair McSplit algorithm described in the previous section. In fact, best-pair McSplit could prioritize nodes with a higher probability of being paired over others, potentially coming extremely close to the optimal solution on its first attempt.

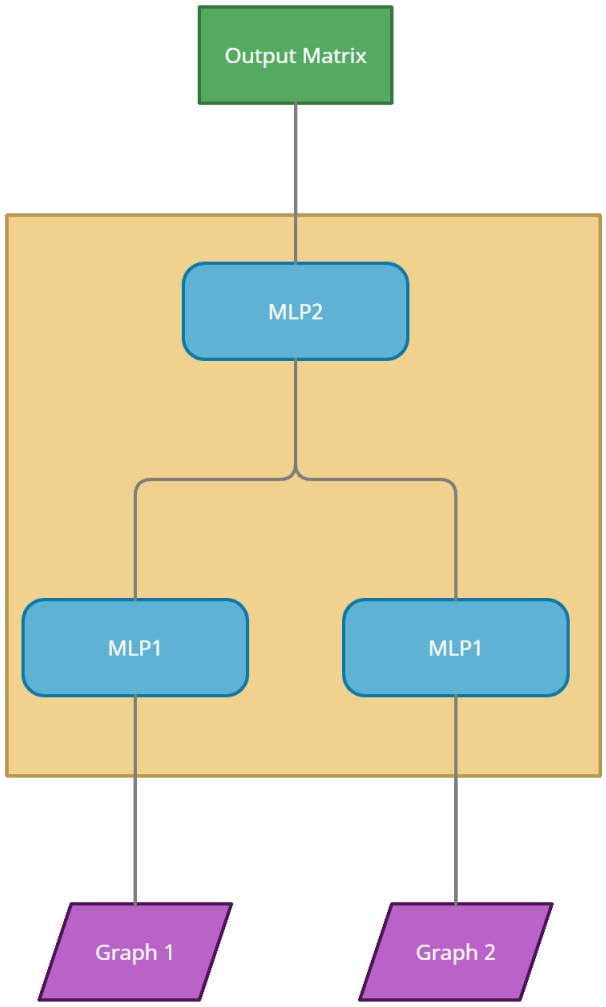


Figure 35: Custom neural network architecture.

Algorithm 6 Python implementation of the 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):
```

5.3 EXPERIMENTAL FINDINGS

5.3.1 *GNN based heuristics*

The implementations of the approaches presented previously (each with multiple settings) are compared to the sequential versions of McSplit [38] McSplit+RL [32].

The implementations are written in C++, the same programming language used by the state-of-the-art, and instead of sorting nodes based on their degree, they load an arbitrary score vector that is then used to pair nodes. The score vectors are precomputed using a Python script that can utilize the Pytorch framework, the Pytorch Geometric library, and the DeepSnap library, which are all used by NeuroMatch [69]. This strategy combines the Python framework's efficient neural network environment with a faster language.

This is significant because the Python implementation of the original algorithm, which is the portion of the process that consumes the most time, is one to two orders of magnitude slower than the original algorithm. The instances were retrieved from the McSplit code repository [38].

The resulting experiments were performed with an i9 9900K CPU and an Nvidia GTX 1060 GPU for embedded computation. The machine runs on Ubuntu 20.04 with a version 2 Windows Subsystem for Linux. The experiments were conducted on two sets of graphs to ensure equality:

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

All runs have a time limit of 50 minutes. In addition, as suggested by [34], a few hyper-parameters are selected to effectively train NeuroMatch. Following their analysis, the proposed NeuroMatch model employs an 8-layer GraphSAGE and a 64-dimensional embedding space. This appears to be the configuration that maximizes NeuroMatch's accuracy, as measured by the number of correct matches. In addition, experiments are conducted with k ranging from 1 to 8. As the number of GNN layers was also fixed at 8, larger values of k would not affect the results, as nodes at distances greater than 8 would be ignored.

The norm heuristic

Table 4 displays the results of the proposed norm heuristic considering all possible values of k for the set of small-sized graph pairs. For these benchmarks, all algorithms are capable of completing the computation within the allotted wall-clock time. The table displays the number of times the proposed version outperformed McSplit and McSplit+RL. Since each method finds the optimal solution, they are compared based on the number of recursion steps they take. These instances are relatively simple to solve, and as demonstrated in section [31], McSplit+RL performs poorly on them.

	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 4: A comparison of the original McSplit heuristic and McSplit+RL against the norm heuristic with varying k values on the smaller graphs.

Table 5 displays the results for instances in which at least one of the methods timed out. It is formatted and presented identically to table 4.

	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 5: Number of victories of the norm-based strategy over the original McSplit heuristic and McSplit+RL on larger graphs.

In this instance, the size of the solution and the number of recursions required to compute it are evaluated to determine the winner. Specifically, the winning method is the one that finds a solution that is larger than the other. Alternately, when two results have the same size, ties are broken by counting the number of recursion steps required to reach the initial solution size. For smaller graphs, the proposed method's advantage decreases with all k values, while for larger graphs, the number of wins oscillates around half the number of graphs. Even if a method can explore the entire search space more quickly, this does not necessarily mean that it can find a better solution in less time. The results of both smaller and larger graphs indicate that the norm heuristic outperforms McSplit in approximately fifty percent of cases for each k value.

Table 6 displays the number of cumulative wins of the proposed algorithm over the original version of McSplit (which, based on the experiments performed, is more efficient than McSplit+RL) on the large graph pair set, given all eight possible values of k. Keeping in mind that there are multiple parallel implementations of McSplit [46], the cumulative result can be easily achieved by running multiple instances of McSplit in parallel, with each version adopting a different value for k. Predicting the optimal value of k for each instance is another way to reduce resource consumption. There may be a correlation between k and the size of the graph, given that k represents the hop "distance" considered from the selected node. To date, it has not been possible to find any correlation between such measures and others.

	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: Cumulative (from left to right) wins of the norm heuristic against the original McSplit heuristic for various k values.

Since k controls the depth at which k-hop neighborhoods are sampled, larger or denser graphs can benefit from greater k values. Consequently, the first step is to search for a correlation between k and graph measures such as graph order, average number of edges, density, or diameter. Unfortunately, it has not been possible to find a correlation between these measures and anything else. Table 7 provides a comparison of the proposed heuristic with norm recomputation, i.e., recalculating the norm after each recursion, to McSplit.

Recalculating the norm at each step is also ineffective, as the algorithm is rendered up to 30,000 times slower without yielding a significant improvement.

An attempt to recompute norms at each recursion step was made. The test was conducted with a k value of 1. Adopting the current implementation, which is in no way optimized, recomputing the norm slows down the algorithm by a factor of 30,000, but it did not outperform McSplit's original heuristic or the proposed norm heuristic. For the evaluation, a fixed number of recursion depths is considered, approximately 5K. The method that finds the largest common subgraph is deemed the winner.

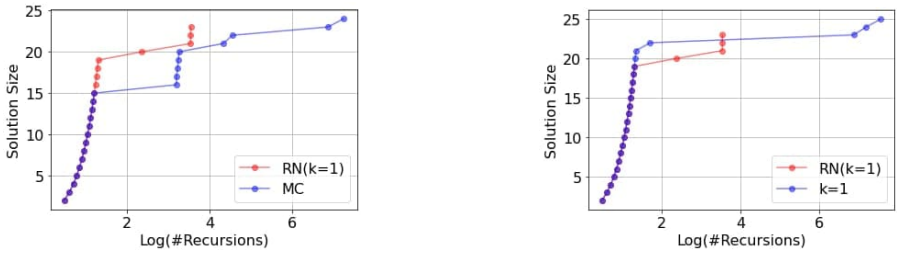
Generally, the McSplit Python re-implementation is capable of reaching 80M recursions in the predetermined 10 minutes

time budget. This occurs with both its original heuristic and the newly proposed one. When trying to re-compute norms at each recursion step, only about 3K recursions can be made, therefore the algorithm is about 30,000x slower. For that reason the table reports the comparison considering an equal number of recursions, around 5K. A decision to to not pursue this line of inquiry was made because norm recomputation is too costly, despite the fact that the two approaches appear to be equivalent even when accounting for the number of recursions.

Baseline	Wins	Losses
k=1	36	34
McSplit	30	40

Table 7: Comparison of the original McSplit heuristic and the proposed norm heuristic without recomputing the norm (with k=1) and the proposed heuristic when recomputing the norm at every recursion step (k still equal to 1).

Figure 36 compares the proposed heuristic while recalculating the norm to the original McSplit heuristic and the proposed heuristic without recomputation for the same problem instance. The norm recomputation heuristic is inferior to both, primarily due to its slow speed. It has an advantage when considering less than 10000 recursions, which is more than three times the total number of recursions the norm recomputation heuristic can perform in 10 minutes.



(a) (b)

Figure 36: Comparison of the norm heuristic proposed with and without norm recomputation. Up to a certain number of recursions, the proposed heuristic with norm recomputation (RN) outperforms McSplit is original heuristic. After that number, norms may lose their effectiveness and the initial heuristic may converge more rapidly.

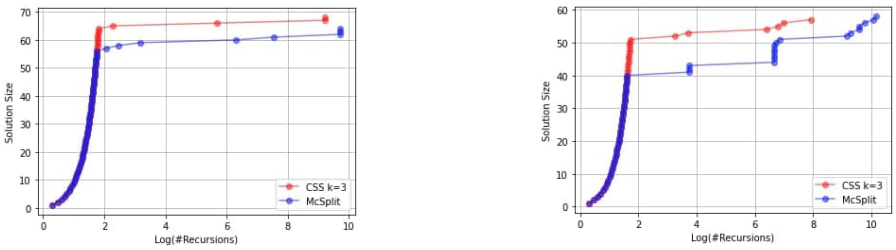
Cumulative Cosine Similarity

Table 8 compares the cumulative cosine similarity heuristic to McSplit and McSplit+RL for the smaller graph pair set. The norm heuristic outperforms the cumulative cosine similarity in this instance. In this method, scores are computed using information from both graphs, as opposed to just one node.

	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 8: The number of victories of the cumulative cosine similarity-based strategy over the McSplit original heuristic and McSplit+RL on small graph pairs.

Figure 37 depicts a detailed analysis of two instances of the problem, of the cumulative cosine similarity heuristic versus McSplit, reporting the solution size (along the y-axis) as a function of the number of recursions (x-axis). The number of recursions (reaching 74B in the first case and 73B in the second) is reported on a logarithmic scale for readability purposes. Each dot on the two graphs represents the first time the corresponding heuristic achieves a particular solution size. Notably, the cumulative cosine similarity algorithm finds larger solutions in less time than the original algorithm.



(a) (b)

Figure 37: The heuristic of cumulative cosine similarity (k=3) in comparison to McSplit is presented here. In a shorter period of time, larger solutions are discovered. First, the proposed heuristic (red) finds a solution with 68 nodes, while the original method (blue, MC) stops at 64 nodes in the same amount of time (left). While McSplit takes 50 minutes to reach nodes, the proposed heuristic in the second case only finds a solution with 57 nodes.

The number of victories achieved by the cumulative cosine similarity heuristic on the large graph pair set is displayed in

table 9. The benefits of this heuristic are readily apparent, as it also outperforms the norm-based approach. This is because of the behavior depicted in figure 37 and described previously. In this case, however, unlike the standard heuristic, results obtained with varying k values are typically not complementary; thus, varying k values cannot be used to generate competing portfolio engines.

	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 9: Number of victories of the cumulative cosine similarity-based strategy over the McSplit original heuristic and McSplit+RL on larger graphs.

Combining the two best values of k for both heuristics, namely $k = 1$ and $k = 2$ for the norm and $k = 3$ and $k = 4$ for the cumulative cosine similarity heuristic, yields the results shown in table 10. The table demonstrates that these strategies outperform the original heuristics in over ninety percent of instances.

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

Table 10: Cumulative (from left to right) results against McSplit when combining the proposed heuristics with the two best values of k.

5.3.2 *Best-pair McSplit Implementation*

This section shows and discusses the tests performed to evaluate the best-pair McSplit implementation (Section 5.2.3). 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 done with node heuristics. Of course, the modified solver is compared with a python re-implementation of McSplit. Tests are performed on 70 instances, randomly selected from the dataset [38] used by McCreesh, Prosser, and Trimble [37].

Note that in table 11, the proposed methods are unable to outperform McSplit. On average, they can reach a relatively

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

Table 11: The two modified versions of McSplit (best-pair on Selected BiDomain (BPBD) and best-pair across all BiDomains (BP)) are superior to the original McSplit.

large solution size. This demonstrates that prioritizing node pairs over single nodes may still be possible using a different heuristic. Cosine similarity is used to prioritize node pairs in this case. Either the embeddings can be improved so that the cosine similarity yields more accurate results, or a different method for prioritizing node pairs should be explored.

6 CONCLUSIONS

In an effort to improve upon the state-of-the-art methods to solve the maximum common subgraph problem, several strategies have been proposed in this dissertation. The experimental findings presented in this work highlight how researching and improving upon existing heuristics can yield significant results. As discussed in chapters 1 and 2, the focus on the maximum common subgraph problem is justified by the generality that it offers with respect to other similarity measures, particularly because it can be extended to more than two graphs. Two types of approaches were proposed:

- Implementing a solution with a completely new architecture that reuses some McSplit components.
- Replacing or improving heuristics used by McSplit in order to obtain a speed up. These exploit the power of *graph neural networks* to match nodes with similar neighborhoods sooner rather than later in the exploration of the solution space.

6.1 ALIKE

Alike is an approximate algorithm that reuses the McSplit bound formula and bidomain data structure, enhancing it by extending its applicability to graph sets rather than pairs. It employs the bound formula to inform its best-first approach and to prune the search space. Alike is designed with parallelism in mind and can be easily modified to perform as an exact solver or, on the opposite end of the spectrum, to be highly resource efficient by employing a greedy and non-exhaustive strategy.

Even though Alike has not been able to compete with state-of-the-art solvers in terms of performance, the features it provides are significant. Some are not available in any other tool, including support for different solution quality measures and the ability to continue searching only a subset of graphs that are more similar to one another when dealing with more than two graphs.

As anticipated, the obtained results have not been as satisfactory; however, the design concepts underlying Alike remain valuable, and it is possible that optimizing the implementation

will result in significant performance improvements across all dimensions.

An important area of development is optimizing the context switch required when a worker thread is rescheduled to focus on a different portion of the search tree. Another area that needs work is memory usage, which becomes unmanageable on tasks with lengthy timeouts because a sizable portion of the search tree is kept in memory throughout the entire execution.

6.2 HEURISTICS BASED ON GNNS

6.2.1 *Node-ordering heuristics*

Section 5.2.2 discusses two node-ordering heuristics that use the node embeddings produced by a graph neural network. NeuroMatch is the graph neural network architecture that embeds the k-hop neighborhood around each node to generate an embedding.

Instead of node degrees, norm and cosine similarity prioritize nodes during branching. Different k-parameter values were tested, and while a definitive advantage over the original heuristic could not be achieved with norms, it performed better in roughly half of the test instances.

A clear advantage could be observed when considering the different orderings obtained with varying values of k. Interestingly, the cumulative cosine similarity heuristic performs better for all values of k. The combination of the best k values both in norms and cumulative cosine similarity outperforms the node-degree heuristic in almost all cases.

Among potential future works, the following are particularly interesting:

- Expand the graph types supported by the proposed method. NeuroMatch can be adapted to work with labeled, directed, and even larger graphs, so this is certainly feasible.
- Find a replacement for the McSplit bidomain heuristic, which currently restricts branching procedure options.
- Test how supervised methods compare on the same task.
- Perform experiments on large graphs to evaluate the scalability of the proposed approaches.

6.2.2 *Best node-pair Heuristic*

A modification to McSplit that directly enables the selection of pairs of nodes has been discussed in section 5.2.3. The heuristic is based on cosine similarity, meaning that node pairs with the highest cosine similarity are chosen first. This intuitively prioritizes the search on node pairs that are more likely to be part of the MCS solution.

The proposed implementation is fully functional, but performance enhancements are possible. Future efforts will focus on increasing the quality of the embeddings, which should result in a more precise cosine similarity measure. Designing new heuristics for ordering node pairs will also be an intriguing area of future study (not necessarily using GNNs).

BIBLIOGRAPHY

[1] *Alike experimental results*. URL: https://docs.google.com/spreadsheets/d/1Fygpr6GeVsCRLsTkCtZzPhI7qdxIgx5D_SSKkQ4HMpk/edit?usp=sharing.

[2] *ARG Graph database*. URL: https:%20//mivia.unisa.it/datasets/graph-database/arg-database/.

[3] Yunsheng Bai et al. "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.

[4] Egon Balas and Chang Sung Yu. "Finding a maximum clique in an arbitrary graph". In: *SIAM Journal on Computing* 15.4 (1986), pp. 1054–1068.

[5] *Beta-D-glucose-3D-balls.png*. 2007. URL: https://en.wikipedia.org/wiki/File:Beta-D-glucose-3D-balls.png.

[6] B. Bollobás et al. *Modern Graph Theory*. Graduate Texts in Mathematics. Springer New York, 1998. ISBN: 9780387984889. URL: https://books.google.com/books?id=SbZKSZ-1qrwC.

[7] H. Bunke. "On a relation between graph edit distance and maximum common subgraph". In: *Pattern Recognition Letters* 18.8 (Aug. 1997), pp. 689–694. DOI: 10.1016/s0167-8655(97)00060-3. URL: https://doi.org/10.1016/s0167-8655(97)00060-3.

[8] Horst Bunke, Xiaoyi Jiang, and Abraham Kandel. "On the minimum common supergraph of two graphs". In: *Computing* 65.1 (2000), pp. 13–25.

[9] Enrico Carraro and Thomas Madeo. *Alike repository*. 2021. URL: https://github.com/enricocarraro/alike.

[10] D. A. Cohen et al. "The Tractability of CSP Classes Defined by Forbidden Patterns". In: *Journal of Artificial Intelligence Research* 45 (Sept. 2012), pp. 47–78. DOI: 10.1613/jair.3651. URL: https://doi.org/10.1613/jair.3651.

[11] David Cohen et al. "Symmetry Definitions for Constraint Satisfaction Problems". In: *Constraints* 11.2-3 (June 2006), pp. 115–137. DOI: 10.1007/s10601-006-8059-8. URL: https://doi.org/10.1007/s10601-006-8059-8.

[12] Stephen A Cook and David G Mitchell. "Finding hard instances of the satisfiability problem: A survey". In: *Satisfiability Problem: Theory and Applications* 35 (1997), pp. 1–17.

[13] Martin C Cooper, Peter G Jeavons, and András Z Salamon. "Generalizing constraint satisfaction on trees: Hybrid tractability and variable elimination". In: *Artificial Intelligence* 174.9-10 (2010), pp. 570–584.

[14] Pierluigi Crescenzi and Alessandro Panconesi. "Completeness in approximation classes". In: *Information and Computation* 93.2 (1991), pp. 241–262.

[15] Rina Dechter and Judea Pearl. "Generalized best-first search strategies and the optimality of A". In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 505–536.

[16] Benjamin S Duran and Patrick L Odell. *Cluster analysis: a survey*. Vol. 100. Springer Science & Business Media, 2013.

[17] Paul J Durand et al. "An efficient algorithm for similarity analysis of molecules". In: *Internet Journal of Chemistry* 2.17 (1999), pp. 1–16.

[18] Hans-Christian Ehrlich and Matthias Rarey. "Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review". In: *WIREs Computational Molecular Science* 1.1 (Jan. 2011), pp. 68–79. DOI: 10.1002/wcms.5. URL: https://doi.org/10.1002/wcms.5.

[19] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979. ISBN: 0716710455.

[20] Aditya Grover and Jure Leskovec. "node2vec". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, Aug. 2016. DOI: 10.1145/2939672.2939754. URL: https://doi.org/10.1145/2939672.2939754.

[21] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 1025–1035. ISBN: 9781510860964. DOI: 10.5555/3294771.3294869. URL: https://dl.acm.org/doi/10.5555/3294771.3294869.

[22] Robert M Haralick and Gordon L Elliott. "Increasing tree search efficiency for constraint satisfaction problems". In: *Artificial intelligence* 14.3 (1980), pp. 263–313.

[23] Ramesh Hariharan et al. "MultiMCS: A Fast Algorithm for the Maximum Common Substructure Problem on Multiple Molecules". In: *Journal of Chemical Information and Modeling* 51.4 (2011). PMID: 21446748, pp. 788–806. DOI: 10.1021/ci100297y. eprint: https://doi.org/10.1021/ci100297y. URL: https://doi.org/10.1021/ci100297y.

[24] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[25] Philippe Jégou. "Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems". In: *AAAI*. Vol. 93. 1993, pp. 731–736.

[26] William Kocay and Donald L Kreher. *Graphs, algorithms, and optimization*. Chapman and Hall/CRC, 2016.

[27] Nils Morten Kriege. "Comparing graphs". PhD thesis. 2015.

[28] Evgeny B Krissinel and Kim Henrick. "Common subgraph isomorphism detection by backtracking search". In: *Software: Practice and Experience* 34.6 (2004), pp. 591–607.

[29] Giorgio Levi. "A note on the derivation of maximal common subgraphs of two directed or undirected graphs". In: *Calcolo* 9.4 (1973), p. 341.

[30] Kevin Leyton-Brown et al. "A portfolio approach to algorithm selection". In: *IJCAI*. Vol. 3. 2003, pp. 1542–1543.

[31] Yanli Liu et al. "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.

[32] Yanli Liu et al. *McSplit+RL code repository*. URL: https://github.com/JHL-HUST/McSplit-RL.

[33] Cardone Lorenzo. "The maximum common subgraph problem: concurrency and multi-graph extentions". In: (2021). URL: https://webthesis.biblio.polito.it/21140/.

[34] Zhaoyu Lou et al. "Neural subgraph matching". In: *arXiv preprint arXiv:2007.03092* (2020). DOI: 10.48550/arXiv.2007.03092. URL: https://doi.org/10.48550/arXiv.2007.03092.

[35] Alan K. Mackworth. "Consistency in networks of relations". In: *Artificial Intelligence* 8.1 (Feb. 1977), pp. 99–118. DOI: 10.1016/0004-3702(77)90007-8. URL: https://doi.org/10.1016/0004-3702(77)90007-8.

[36] Ciaran McCreesh. "Solving hard subgraph problems in parallel". PhD thesis. University of Glasgow, 2017.

[37] Ciaran McCreesh, Patrick Prosser, and James Trimble. "A partitioning algorithm for maximum common subgraph problems". In: (2017).

[38] Ciaran McCreesh, Patrick Prosser, and James Trimble. *McSplit code repository*. 2017. URL: https://www.github.com/jamestrimble/ijcai2017-partitioning-common-subgraph.

[39] Ciaran McCreesh, Patrick Prosser, and James Trimble. "The Glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants". In: *International Conference on Graph Transformation*. Springer. 2020, pp. 316–324.

[40] James J McGregor. "Backtrack search algorithms and the maximal common subgraph problem". In: *Software: Practice and Experience* 12.1 (1982), pp. 23–34.

[41] Samba Ndojh Ndiaye and Christine Solnon. "CP Models for Maximum Common Subgraph Problems". In: *Principles and Practice of Constraint Programming – CP 2011*. Springer Berlin Heidelberg, 2011, pp. 637–644. DOI: 10.1007/978-3-642-23786-7_48. URL: https://doi.org/10.1007/978-3-642-23786-7_48.

[42] Andreas Papadopoulos, George Pallis, and Marios D. Dikaiakos. "Identifying Clusters with Attribute Homogeneity and Similar Connectivity in Information Networks". In: *2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)*. IEEE, Nov. 2013. DOI: 10.1109/wi-iat.2013.49. URL: https://doi.org/10.1109/wi-iat.2013.49.

[43] Marcello Pelillo. "Replicator Equations, Maximal Cliques, and Graph Isomorphism". In: *Neural Computation* 11.8 (Nov. 1999), pp. 1933–1955. DOI: 10.1162/089976699300016034. URL: https://doi.org/10.1162/089976699300016034.

[44] 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. DOI: 10.1145/2623330.2623732. URL: https://doi.org/10.1145/2623330.2623732.

[45] Thierry Petit, Jean-Charles Régin, and Christian Bessière. "Specific Filtering Algorithms for Over-Constrained Problems". In: *Principles and Practice of Constraint Programming — CP 2001*. Springer Berlin Heidelberg, 2001, pp. 451–463. DOI: 10.1007/3-540-45578-7_31. URL: https://doi.org/10.1007/3-540-45578-7_31.

[46] Stefano Quer, Andrea Marcelli, and Giovanni Squillero. "The Maximum Common Subgraph Problem: A Parallel and Multi-Engine Approach". In: *Computation* 8.2 (May 2020), p. 48. DOI: 10.3390/computation8020048. URL: https://doi.org/10.3390/computation8020048.

[47] Stefano Quer and Gabriele Mosca. "Solving the maximum common subgraph problem on many-cores architectures". In: (2019). URL: https://webthesis.biblio.polito.it/11038/1/tesi.pdf.

[48] John W Raymond and Peter Willett. "Maximum common subgraph isomorphism algorithms for the matching of chemical structures". In: *Journal of computer-aided molecular design* 16.7 (2002), pp. 521–533.

[49] Jean-Charles Régin. "A filtering algorithm for constraints of difference in CSPs". In: *AAAI*. Vol. 94. 1994, pp. 362–367.

[50] "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.

[51] John R Rice. "The algorithm selection problem". In: *Advances in computers*. Vol. 15. Elsevier, 1976, pp. 65–118.

[52] Francesca Rossi, Peter Van Beek, and Toby Walsh. "Constraint programming". In: *Foundations of Artificial Intelligence* 3 (2008), pp. 181–211.

[53] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[54] Daniel Sabin and Eugene C. Freuder. "Contradicting conventional wisdom in constraint satisfaction". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1994, pp. 10–20. DOI: 10.1007/3-540-58601-6_86. URL: https://doi.org/10.1007/3-540-58601-6_86.

[55] Christian Schulte and Peter J Stuckey. "Efficient constraint propagation engines". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31.1 (2008), pp. 1–43.

[56] Christian Schulte and Guido Tack. "Weakly monotonic propagators". In: *International conference on principles and practice of constraint programming*. Springer. 2009, pp. 723–730.

[57] Claude E Shannon. "Prediction and entropy of printed English". In: *Bell system technical journal* 30.1 (1951), pp. 50–64.

[58] Nino Shervashidze et al. "Weisfeiler-Lehman Graph Kernels". In: *Journal of Machine Learning Research* 12.77 (2011), pp. 2539–2561. URL: http://jmlr.org/papers/v12/shervashidze11a.html.

[59] SG Shirinivas, S Vetrivel, and NM Elango. "Applications of graph theory in computer science an overview". In: *International journal of engineering science and technology* 2.9 (2010), pp. 4610–4621.

[60] Giovanni Squillero, Stefano Quer, and Mattia De Prisco. "An Approximate Graph-Similarity Algorithm based on Monte Carlo Tree Search". In: ().

[61] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435. DOI: 10.5555/2627435.2670313. URL: https://dl.acm.org/doi/10.5555/2627435.2670313.

[62] Madeo Thomas. "Graph neural networks for the MCS problem". In: (2022). URL: https://webthesis.biblio.polito.it/22683/1/tesi.pdf.

[63] Jacobo Torán. "On the Hardness of Graph Isomorphism". In: *SIAM Journal on Computing* 33.5 (Jan. 2004), pp. 1093–1108. DOI: 10.1137/s009753970241096x. URL: https://doi.org/10.1137/s009753970241096x.

[64] J. R. Ullmann. "An Algorithm for Subgraph Isomorphism". In: *Journal of the ACM* 23.1 (Jan. 1976), pp. 31–42. DOI: 10.1145/321921.321925. URL: https://doi.org/10.1145/321921.321925.

[65] Petar Veličković et al. *Graph Attention Networks*. 2017. DOI: 10.48550/ARXIV.1710.10903. URL: https://arxiv.org/abs/1710.10903.

[66] Philippe Vismara and Benoît 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.

[67] Hui Wei, Chengzhuan Yang, and Qian Yu. "Efficient graph-based search for object detection". In: *Information Sciences* 385-386 (Apr. 2017), pp. 395–414. DOI: 10.1016/j.ins.2016.12.039. URL: https://doi.org/10.1016/j.ins.2016.12.039.

[68] Keyulu Xu et al. "How powerful are graph neural networks?" In: *arXiv preprint arXiv:1810.00826* (2018). DOI: 10.48550/arXiv.1810.00826. URL: https://doi.org/10.48550/arXiv.1810.00826.

[69] Rex Ying et al. *Neural Subgraph Matching*. [Online; accessed September, 2021]. 2020. URL: http://snap.stanford.edu/subgraph-matching/files/neuromatch_arch.png.

[70] Zhiping Zeng et al. “Comparing stars: On approximating graph edit distance”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 25–36.

[71] Si Zhang et al. “Graph convolutional networks: a comprehensive review”. In: *Computational Social Networks* 6.1 (2019), pp. 1–23. DOI: 10.1186/s40649-019-0069-y. URL: https://doi.org/10.1186/s40649-019-0069-y.