POLITECNICO DI TORINO

Master's Degree in Computer Engineering
Artificial Intelligence and Data Analytics

Master's Degree Thesis

Graph Neural Networks, Multi-Threading and Heuristics for the Computation of the Maximum Common Subgraph

Supervisors

Prof. Stefano QUER

Prof. Giovanni SQUILLERO

Dr. Andrea CALABRESE

Candidate

Salvatore LICATA

Matr. s295798

A. A. 2022/2023

Abstract

In recent years, we have seen a rising usage of graph-like data structures in several fields, with a wide range of applications in real-world scenarios. Various problems known by the computer science community for a long time have come back to the surface, backed up by new ideas and algorithms to challenge their commonly high degree of complexity.

The Maximum Common Subgraph problem is one of them, as it is a well-known computational problem in graph theory. The MCS of the two graphs is the largest common subgraph between them, and it has great relevance in numerous fields, ranging from computer vision to malware detection and bioinformatics. Unfortunately, this problem is NP complex, making it difficult to solve in a reasonable amount of time. Many researchers have tried to develop novel approaches to explore the enormous search space generated on average by this problem.

Since 2016, a new algorithm, McSplit, has become one of the most commonly adopted baselines to face this task. McSplit is a recursive procedure that exploits a branch-and-bound technique to prune the search space aggressively when the current evaluation tree is upper-bounded. Each recursion step chooses which vertex pair from the two graphs has to be added to the current solution through the score returned by a heuristic function. This strategy laid the foundation for many subsequent works, which mainly aimed at enhancing it by using more efficient heuristics. Other approaches have also leveraged novel Deep Learning methods, like Graph Neural Networks, exploiting big data to train models to find deep representations inside graphs. This work explores three different techniques to extend the current state-of-the-art McSplit variant, called McSplitDAL.

In the first one, we analyze the most recent variants of McSplit: McSplitRL, McSplitLL, and McSplitDAL. These algorithms maintain the same core as the original one but substitute the scoring heuristic with a table of rewards, accurately updated as in classical Reinforcement Learning frameworks. We also introduce new heuristics, which proved to lead to better solutions than the out-of-the-box variants of these algorithms.

In the second one, we focus on exploiting parallelization. We build an iterative version of the McSplit algorithm to optimize the load distribution among different

threads and allow for a faster and broader search space exploration. Additionally, we introduce an enhancement of an existing parallel version of McSplit using our heuristics, which provides benefits with almost no additional overhead.

In the last one, we present techniques based on Graph Neural Networks. This methodology differs completely from the previous ones, and it has been inspired by the recent growing interest in GNNs. We propose two different GNN-based trainable architectures to select the vertices to be added to the current MCS at each algorithm step.

Concerning results, the first approach improves the current state-of-the-art on single-thread execution, and it has been accepted at ICSOFT 2023 (18th International Conference on Software Technologies) as a full paper. The parallel strategies outperform the single-thread algorithm, proving the benefits of parallelization. Finally, the GNN-based technique obtains good results, and it is able to process larger graphs. Among the proposed methods, the last one has the most significant chance of improvement: accurate finetuning and a more complex architecture may provide additional gains.

Acknowledgements

I desire to express my appreciation to all who have supported me throughout my academic path, which concludes with the end of this master's thesis.

I thank my supervisors, Stefano Quer, Giovanni Squillero, and Andrea Calabrese, which guided me throughout this work.

I am grateful to my dad Pietro, my mum Teresa and my sister Giorgia, who always supported me, to Laura Aurora, who has always been by my side, and to the rest of my family, with a special dedication to my grandparents, who always rooted for me.

Last but not least, I want to thank my dearest friends who always believed in me.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Graphs are data structures that, through a mathematical abstraction, provide great flexibility for modeling entities and analyzing the interactions between them. They are widely used across various fields, such as computer science, mathematics, chemistry, and engineering, to name a few.

Graph theory, the mathematical study of graphs, provides a robust framework for understanding and solving many complex practical tasks. Among these tasks, we focus on the Maximum Common Subgraph (MCS) problem.

The MCS problem involves finding the largest subgraph common to two given graphs. The MCS has applications in various fields, such as bioinformatics, image recognition, and pattern matching. By exploiting graph theory concepts and algorithms, researchers can devise efficient strategies to tackle the MCS problem and extract meaningful information from complex data sets. The rich theoretical background of graph theory enhances our understanding of the MCS problem and enables us to develop efficient solutions.

In recent years, a novel approach to solving the MCS problem has become a widespread standard: McSplit, an algorithm introduced by McCreesh et al. ([1]). This approach has been the foundational block when it comes to MCS and many authors in the past years have tried to improve it, to achieve better performances.

In this thesis, we tackle a more restricted class of the MCS problem, the so-called Maximum Common Induced Subgraph (MCIS), which adds additional constraints on the resulting common subgraph (more details later). We tried to solve this problem using different techniques, specifically Heuristics, Reinforcement Learning (RL), Multi-Threading and Graph Neural Networks (GNN).

This work is articulated as follows. Chapter 2 provides detailed theoretical background on graphs, the MCS problem, the McSplit algorithm, Reinforcement Learning, and GNNs. These notions will be necessary for understanding the following chapters.

Chapter 3 focuses on our first approach, which exploits novel heuristics and

more advanced versions of McSplit to improve the original algorithm.

Chapter 4 approaches the problem by exploiting parallelism, providing a new version of the McSplit algorithm suitable for parallelization and enhancing an existing one.

Chapter 5 analyzes the current state-of-the-art approach called GLSearch ([2]) and introduces two simple yet suitable methods for solving the MCS problem using Graph Neural Networks.

Finally, in Chapter 6, we draw some conclusions on the previously treated methods and propose future works.

Chapter 2

Theoretical background

This chapter introduces all the required common knowledge for understanding our contributions, presented later.

More specifically, section 2.1 explains graphs in detail since we based our entire work on this data structure.

Section 2.2 treats the Maximum Common Subgraph problem and discusses the different variants, specifically the induced and the connected ones. The former is the one considered in our experiments, while the latter is the one tackled by the authors of GLSearch ([2]), discussed in Chapter 5.

Section 2.3 explains in detail the McSplit algorithm ([1]) since it is the baseline of all the methods explored in the subsequent chapters and it provides common data structures which we extensively used throughout this thesis.

Section 2.4 talks about Reinforcement Learning because many recent variants of the McSplit algorithm, discussed in Chapter 3, make use of this approach for choosing the vertices of the resulting MCS. Moreover, GLSearch utilizes Deep Q Networks (DQN), a Reinforcement Learning technique.

Finally, section 2.5 explores Graph Neural Networks, a field of Deep Learning which combines the power of graphs with neural networks. It is a fast-growing research area that has experienced a boost in recent years and will probably continue like this in years to come.

2.1 Graphs

A graph G is a data structure made by a set of vertices V and the set of connections among them, the edges E. We denote a graph as $G = \{V, E\}$. A *simple graph* does not contain self-loops (a vertex linked to itself) or multiple edges between the same pair of vertices.

Graphs are classified based on the properties of their vertices, edges, or both.

They can be *labeled* if vertices contain extra information (a label) or *unlabeled* if they have no additional data.

A graph is said to be *weighted* if it assigns numerical values, called weights, to the edges to represent a property associated with the relationship (for example, in routing, the cost of that connection). If there are no weights, the graph is called *unweighted*, so it is like a weighted one with all weights equal to 1.

Lastly, graphs can be *directed* or *undirected*. In an undirected graph, the edges have no specific direction, and the relationship between two vertices is symmetric. In contrast, in a directed graph (or *digraph*), the edges have a specific direction, indicating a one-way relationship between the vertices. Formally, we say that G is *undirected* if:

$$\forall v_1, v_2 \in V(G) \in E(G) \iff \{v_2, v_1\} \in E(G) \wedge E(v_1, v_2) = E(v_2, v_1)$$

Figure 2.1 is an example of a directed graph, while Figure 2.2 shows an undirected one.

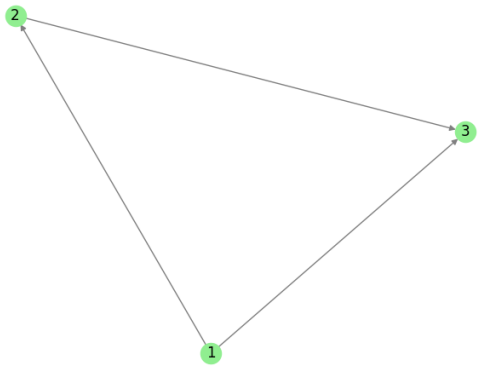

Figure 2.1: Directed graph

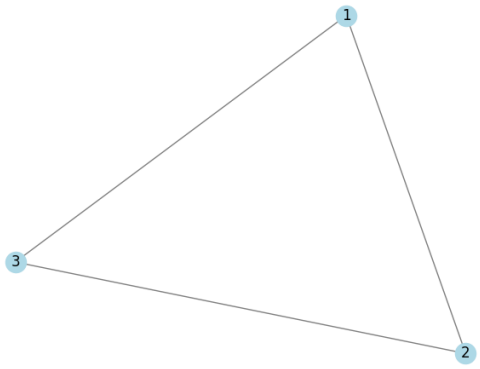

Figure 2.2: Undirected graph

Our work focuses on undirected, unlabeled, and unweighted graphs, which can also contain self-loops. This choice was mainly due because it represents the worst case, in terms of complexity, for the MCS problem, explained in section 2.2.

2.1.1 Representations

Graphs have different representations depending on the specific requirements and algorithms used. The most common representation methods are the *adjacency matrix* and the *adjacency list*.

An adjacency matrix is a square matrix of size $N \times N$ where $N = |V(G)|$, the cardinality of the set of vertices. Each row and column correspond to a vertex,

and the entry at position (i, j) represents the connection between vertices i and j. This representation is efficient for accessing information but wastes too many resources for memory allocation, especially when the graph has low connectivity (low number of edges, sparse matrix).

In contrast, an adjacency list uses dynamic allocation to represent the connections of each vertex. Each vertex has a list containing its adjacent vertices. This strategy is more expensive in terms of time complexity for accessing the information about the connection of two vertices, but it is very efficient in terms of memory since it allows the allocation of only the neighbors of each vertex.

2.1.2 Definitions

In this subsection, we consider two graphs, G and H, and we provide some definitions which will be helpful later.

We say that G and H are isomorphic if we can identify a bijection between the two graphs such that:

$$\forall v_1, v_2 \in H \in E(H) \iff \{v_1, v_2\} \in E(G)$$

In other terms, two graphs are isomorphic if we can map each vertex of G to each vertex of H in a way that the number of edges joining any two vertices of G is the same as the number of edges joining the corresponding vertices of H.

We define H as a subgraph of G if:

$$V(H) \subset V(G) \wedge E(H) \subset E(G)$$

In other words, H is a subgraph of G if its vertices and edges are a subset of the vertices and edges of G. A more restrictive type of subgraph is the *induced* subgraph, which imposes that all the edges present in G between the nodes of H are also edges of H.

2.2 Maximum Common Subgraph

In graph theory, the Maximum Common Subgraph problem and its induced variant (Maximum Common Induced Subgraph) are known optimization problems. The main goal of these algorithms is finding the largest common subgraph of two given input graphs, which should also be *induced* in the MCIS.

This class of problems is known to be NP-hard and almost intractable in terms of full search due to the exponentially high number of possible combinations. Regardless, both are widely used in many other fields, like computer vision and bioinformatics, as tools for finding large common patterns and analyzing similarities

and differences efficiently, thus enforcing the importance of finding efficient solutions to these tasks.

We can formalize the Maximum Common Subgraph problem as follows. Given two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$, we can define:

$$MCS(G, H) = (V_{MCS}, E_{MCS})$$

such that

- $V_{MCS} \subseteq V_G$
- $V_{MCS} \subseteq V_H$
- $E_{MCS} \subseteq E_G$
- $E_{MCS} \subseteq E_H$
- $MCS(G, H)$ is isomorphic to a subgraph of both G and H
- $|V_{MCS}|$ is maximized

Analogously, the MCIS variant has the same requirements but $MCS(G, H)$ should be isomorphic to an *induced* subgraph of G and H.

There is also an additional version, which we will briefly see in Chapter 5 when talking about GLSearch, which requires the found MCS to be connected, too. In this work, we relax this constraint, allowing it to be disconnected.

2.3 McSplit

In 2017, McCreesh et al. ([1]) introduced a new exact algorithm called McSplit aimed at solving the MCS problem. This algorithm has since been the standard baseline for this task, built upon three core ideas:

- Label classes, to efficiently store the mapping between the current best MCS
- Branch and Bound approach, to aggressively prune the search space
- Heuristics, for choosing the next actions smartly

In the following subsections, we first show a brief overview of the algorithm. Then we analyze separately the three basic blocks we listed above.

2.3.1 Overview

McSplit, as described in Algorithm 1, is a depth-first recursive search that starts with an empty mapping. At each recursion level, it selects a label class between the ones available and pairs two vertices belonging to that label class, one from G and one from H, the input graphs. The solution cannot contain the same vertex multiple times, and matched vertices must share the same label class to build a mapping that will represent a valid MCIS (more details in subsection 2.3.2).

After matching two vertices, the algorithm generates the new label classes according to the current mapping and the new chosen pair, allowing it to pair vertices correctly in the next recursion steps. Each time the current mapping's size exceeds the best one found up to that moment, the *incumbent* (the best mapping) is updated.

The algorithm also allows pruning the search if the bound calculated on the current mapping and the selected label classes are lower than or equal to the size of the incumbent: this allows us to avoid searching unpromising branches. Moreover, the algorithm backtracks and tries different pairings whenever it hits the bound.

It is worth mentioning that this algorithm is exact, so it will eventually find the best solution if it has enough time.

Now we analyze in details the main concepts of this technique.

2.3.2 Label classes

As we said previously, since we need the resulting mapping to be a valid MCIS, we cannot randomly pair a vertex from G with a vertex from W. To solve this issue efficiently, the authors built a labeling method that maps each vertex to a string of binary values. Each time we create a new pair, we select two vertices sharing the same label class, one for each input graph. After this selection, all the remaining vertices add a new character at the end of their label: 1 if they are connected to the vertex selected, 0 if they are not (adjacency is checked against vertices belonging to the same starting graph, obviously). Vertices of G belonging to a label class can thus be mapped to vertices of H until none of the two label classes is empty.

If any of the two label classes belonging to the starting graphs is smaller than the other one at a certain point, we are sure there will be some unmatched vertices inside that label class. This last remark is the starting point for calculating a solid bound during the search.

2.3.3 Branch and Bound

A search space that grows exponentially due to all possible pairing is completely unmanageable if approached with brute force. The key point in finding a good

Algorithm 1 Pseudocode of the original McSplit algorithm introduced by McCreesh

```
1: Best ← ∅
2:
3: function MCS(G, H, M)
4:     if |M| > |Best| then
5:         Best ← M
6:     end if
7:
8:     if CalculateBound() < |Best| then
9:         return
10:    end if
11:
12:    label_class ← SelectLabelClass(G, H)
13:    v ← SelectVertex(G, label_class)
14:    G′ ← G′ \ {v}
15:
16:    for all w ∈ getVertices(H, label_class) do
17:        M′ ← M ∪ (v, w)
18:        H′ ← H \ {w}
19:        G′ ← UpdateLabels(G′, v)
20:        H′ ← UpdateLabels(H′, w)
21:        MCS(G′, H′, M′)
22:    end for
23:    MCS(G′, H, M)
24:    return
25: end function
```

solution in a small amount of time is not only provided by a smart choice of the vertices to be paired but also by an efficient search pruning.

$$B = |M| + \sum_{l \in L} \min(|\{v \in G \backslash M : L(v) = l\}|, |\{w \in H \backslash M : L(w) = l\}|) \quad (2.1)$$

Equation 2.1 shows the mathematical formula for computing the upper bound of the current search tree. The current mapping and the available label classes influence this bound. As we highlighted in the previous section, the smallest label class of the two input graphs limits the number of possible pairs. Therefore, the current search tree can achieve a mapping whose size is at most the size of the mapping up to that moment plus the sum of the minimum size of all label classes, where the minimum size of a label class is the smaller number of vertices belonging either to G or to H having that label class.

It is possible to compare this bound with the size of the best mapping found up to that moment. If the bound is smaller than or equal to it, the algorithm prunes the current search tree since it will not lead to a mapping bigger than the current best.

2.3.4 Heuristics

At each recursion level, the presented algorithm selects a label class L and a vertex V of G within L. After that, McSplit tries to pair all vertices w of H belonging to L with the selected v and recurs each time it builds a new pair. Two ad-hoc heuristic functions manage the selection of the label class L and of the vertex v.

When it concerns the choice of the label class, the algorithm selects the one having the smallest size, i.e. the one with the minimum number of vertices from G or H. If a tie occurs, the label class containing a vertex of G with the highest degree is selected. This approach first explores smaller label classes, which lead to simplifying the task and to easy solutions, which can then be used as a bound, as discussed in section 2.1.

The selection of v, instead, is based on its degree. The higher the size of the neighborhood of a vertex, the higher the chance of being selected within the same label class. We can explain this by observing that a vertex linked with many other vertices will probably be part of the final MCIS. While this approach is overall good, many authors have tried to improve these specific heuristics. We will add more details in Chapter 3.

2.4 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of Machine Learning aimed at creating programs called agents that can learn to make decisions within a defined

environment maximizing the reward they receive. The main idea behind RL is the trial-and-error learning process of humans and animals. This approach has significantly attracted larger attention in recent years due to its ability to solve complex tasks without requiring a custom program suited for the task.

The agent interacts with the environment, takes actions, and receives feedback as rewards (which can also be negative, hence being penalties). The agent's goal is to learn a *policy* to identify which action has to be taken from its current state to get the maximum cumulative reward in the long term. During the learning phase, exploration and exploitation must be balanced by choosing unknown actions or exploiting current knowledge.

2.4.1 Key components

We can distinguish seven main components in RL:

1. Agent: it is the subject that takes actions and interacts with the environment based on its current state, receiving rewards that can act as positive or negative feedback
2. Environment: the context within the agent operates. It changes its state based on the agent's actions and is responsible for providing rewards
3. State: it represents the environment at a fixed time. It is relevant for the agent since it uses the state to decide the next action
4. Action: it is the interaction the agent makes with the environment based on the current state and which will result in a reward
5. Reward: the feedback the environment provides for a given action taken by the agent. It can be positive or negative, depending on the impact of the action
6. Policy: it is the strategy the agent builds mapping states to actions. It can be deterministic (mapping a specific action to a state) or stochastic (using probability distributions of taking an action for a given state)
7. Value Function: the evaluator the agent uses to assess the long-term reward using a given policy. It is necessary for decision-making and guides the agent in choosing actions

2.4.2 Algorithms

There are many RL algorithms, each with pros, cons, and a more suitable application domain. Among this broad class of algorithms, the ones we will consider in our work are Q-Learning and Deep Q-Learning.

Q-Learning is a value-based RL algorithm that learns an action-value function called Q-function. The agent waits for feedback from the environment and updates the Q-values, which map an environment state to a given action, maximizing rewards received in the long term. This approach is the core idea of McSplitRL, a McSplit extension we will analyze in Chapter 3.

Deep Q-Learning is a more sophisticated variant of Q-Learning, called Deep Q-Networks ([3]), since it uses deep neural networks to approximate the Q-function. It has been successful in solving complex problems by combining the power of Deep Learning and Reinforcement Learning, and it is the fundamental block of GLSearch, a new approach to solving the MCS problem, which we will see in Chapter 5.

2.5 Graph Neural Networks

Graph Neural Networks (GNNs) are a recently introduced class of deep learning models which operates on graph-like data structures. Unlike standard deep learning models, GNNs can capture relationships inside graphs and express them in compact forms without using other representations. This feature makes GNNs suitable for tasks like node classification, graph classification, link prediction, and efficient graph (or neighborhood) representation.

2.5.1 Main features

GNNs can leverage graph attributes, both local and global, to express connections between nodes and learn a custom representation of the underlying graph ([4]).

Another feature of GNNs is end-to-end learning from raw graph data: this is incredibly convenient because it removes the need for manual feature engineering and graph conversions since it can handle various graph formats of variable size.

A great advantage that is common practice in the field of Deep Learning is transferability, the ability to train a model on a domain and apply it to another one without additional training. It is also possible with GNNs, which can thus be very flexible and easily generalizable.

Finally, one of the most crucial characteristics of GNNs: scalability. Recently, new techniques have allowed GNNs to scale efficiently and handle large graphs, which is necessary since public datasets often come from real-world scenarios where the size cannot be easily limited.

2.5.2 Key components

GNNs are models built around some core concepts: feature representations (at node, edge, or graph level), message passing, convolutional layers, and pooling layers.

Node representations allow us to efficiently encode each vertex of the graph into a feature vector containing its attributes. This vector can include information about the node, which can be of different types (numerical, categorical, or even textual).

Edge representations focus on capturing relationships and interactions among nodes. They can also be weighted, thus encoding the importance of the connection.

Graph representations aim at gathering global information, like the structure, the number of nodes and edges, the type of the graph, et cetera.

Message Passing is the strategy to pass information inside a GNN. It is a fundamental part since it would not otherwise be possible to encode interactions between distant nodes which are not linked directly. This approach usually exploits neighbors to carry information and propagate it to target nodes which can then update their representations.

Graph Convolutional Layers are the core building blocks of GNNs. Similarly to the convolutional layer in traditional deep learning models, they leverage the neighborhood aggregation and message-passing mechanisms to update node representations based on their local patterns, allowing them to model non-linear interactions and describe complex relationships inside graphs ([5]).

Pooling is a well-known strategy in deep learning. It is generally used after multiple graph convolutional layers to aggregate information and reduce the representation size while keeping learned features, being very useful for tasks like graph classification since it can condense representation easily.

Chapter 3

McSplit and Heuristics

This chapter describes the first contribution included in this thesis. First, we analyze the most promising recent McSplit enhancements and a set of new heuristics which can substitute the degree heuristic proposed by McSplit. After that, we present our experiments and the results, showcasing how we can improve the current state-of-the-art McSplit variants by combining multiple techniques.

3.1 McSplitRL

McSplitRL is a recent variant of McSplit proposed by Liu et al. ([6]), which exploits Reinforcement Learning to enhance the standard version. The authors' idea is to store two vectors for the rewards: one for the vertices of G and one for the vertices of H. Like in the original version, the algorithm first sorts all vertices according to the Node Degree heuristic, but when it has to choose a new vertex, it selects the one with the highest reward inside the label class.

The scoring system adopted by the authors works as follows: given the set of label classes at a given point of the search, E_v, and the subsequent set of label classes, E'_v, generated by adding a new pair of vertices to the current solution, the reduction of the size of the label classes due to this action is calculated. We can formalize it as in Equation 3.1.

Equation 3.1:

$$\begin{aligned} R(v,w) \quad = \quad & \textstyle\sum_{(V_l,V_r)\in E_v} \min(|V_l|,|V_r|) - \\ & \textstyle\sum_{(V'_l,V'_r)\in E_{v'}} \min(|V'_l|,|V'_r|) \end{aligned} \tag{3.1}$$

Therefore, a couple of vertices (v, w) are given a higher reward if the new label classes, created after adding the pair to the solution, are smaller than before this action: the higher the size reduction, the higher reward.

It is worth noticing that this approach also causes a higher diminution of the bound that McSplit exploits to perform pruning. If the label classes decrease in size, also the bound will decrease, thus reaching the upper bound in fewer steps.

3.2 McSplitLL

This variant, created by Zhou et al. ([7]), is built on McSplitRL, further refining the Reinforcement Learning policy and adding a technique for matching leaves easily. These two methods are Long Short Memory (LSM) and Leaf Vertex Match Union (LUM).

3.2.1 Long Short Memory

LSM modifies the Reinforcement Learning policy adopted by McSplitRL in two ways: using a matrix for vertices belonging to H instead of a single vector and introducing a decay after reaching some thresholds.

Using a matrix for the rewards of the second input graph allows differentiating when a given w vertex is matched with different v vertices, adding a dependency on the choice of the left vertex of the pair.

On the other side, the decay avoids too big rewards, which could cause one to choose multiple times some vertices. Since the scoring system is now asymmetric due to the different structures (vector and matrix, respectively), also the thresholds are different, with a higher one assigned to the right rewards. Whenever the rewards exceed these limits, the algorithm halves them.

3.2.2 Leaf Vertex Union Match

Leaf nodes are vertices that have only one adjacent vertex. Since the MCIS problem requires finding an induced subgraph, all edges between the included nodes in the original graphs must be present in the resulting subgraph. Then, when we pair two vertices, if both have some leaves among their neighbors, we can include all the pairs of leaves we can create without having to perform an entire recursion step since there are no other connections except the one with the current vertex. This strategy can improve the execution speed without having any negative impacts.

3.3 McSplitDAL

McSplitLL already provided a nice improvement, but Liu et al. introduced McSplitDAL ([8]), which further optimize McSplit algorithm. This algorithm has two core ideas: an additional contribution to the reward formula and a hybrid branching

policy. These ideas are discussed in the following dedicated sections, while in the last one, we will analyze our implementation of McSplitDAL since, differently from the other variants, the authors did not provide a public implementation. Therefore, we made ours following the authors' ideas and experimenting with different parameters.

3.3.1 Domain Action Learning

Domain Action Learning (DAL)'s core idea remarks that the size reduction of label classes is not the only important factor because it is also necessary to consider the number of label classes. Therefore, when two actions lead to the same reduction in the size of the generated label classes, the one creating more label classes is preferred since it simplifies the problem. It can be achieved by adding a term, resulting in the formula in Equation 3.2 for assigning a reward.

$$\begin{aligned} R(v,w) \quad = \quad & \textstyle\sum_{(V_l,V_r)\in E_v} \min(|V_l|,|V_r|) - \\ & \textstyle\sum_{(V'_l,V'_r)\in E_{v'}} \min(|V'_l|,|V'_r|) + \\ & |E_{v'}| \end{aligned} \tag{3.2}$$

3.3.2 Hybrid Branching Policy

DAL's contribution deals with specific corner cases, refining the scoring system. However, the authors of McSplitDAL believe the new versions of McSplit, based on Reinforcement Learning, suffer from a possible *Matthew effect*, meaning they can get stuck in a local optimum, continuing to branch on the same subset of vertices, hence restricting the search space.

They propose an alternating approach to overcome this problem by combining McSplitRL policies and McSplitDAL policies. The algorithm starts working as a standard McSplitRL without LSM and DAL. It continues to use this approach until it reaches a threshold of iterations without improvements. After that, the algorithm switches from McSplitRL to McSplitDAL and resets the counter of wasted iterations. This alternating approach can occur many times during a single run and seems to be beneficial, allowing one to avoid local optimum by dynamically changing the strategy for rewarding and selecting vertices.

3.3.3 Implementation

As we previously said, since there is no public implementation of McSplitDAL, we created our version starting from McSplitLL. Following the authors' leads, we introduced the DAL value function and the hybrid branching policy. We kept the thresholds suggested by the original papers for LSM and the bound of iterations

without improvement, leading to the policy switch. We implemented the algorithm in C++, more details will follow in section 3.6.

3.4 McSplitSwap

As we saw in the previous chapter, McSplit works on two input graphs, G and H, but treats them asymmetrically since it first chooses a vertex from G and then maps it to vertices belonging to H. This fixed order could potentially mean $MCS(G,H) \neq MCS(H,G)$ if we cannot run the McSplit to completion due to time constraints. Hence, it would be interesting to see if there is a smart way to choose which of the two input graphs will be the first one, G, and which will be H.

Recently, a researcher from the same group of McCreesh, Trimble, published three possible solutions to this problem ([9]): McSplitSD, McSplitSO, and their generalized form, McSplit2S.

3.4.1 McSplitSD

McSplitSD swaps G with H if H is denser. We can evaluate the density of a generic graph K as in Equation 3.3.

$$K(G) \quad = \quad \frac{|E(G)|}{|V(G)|\cdot(|V(G)|-1)} \tag{3.3}$$

We can thus define the *density extremeness* of a graph as how much its density is close to 0 or 1, according to Equation 3.4.

$$d_e(G) \quad = \quad |\tfrac{1}{2} - K(G)| \tag{3.4}$$

The swap of G and H occurs when the following inequality about their density extremeness holds:

$$d_e(G) \quad > \quad d_e(H)$$

3.4.2 McSplitSO

McSplitSO also performs a simple swap before starting the standard McSplit algorithm, but the switching condition is based on the *order* of the graphs, the number of vertices inside them, preferring as the first graph the smallest of the two. Therefore, it performs the swap when the following inequality holds:

$$n_G \quad > \quad n_H$$

This approach, according to the author, provides a minor improvement to McSplitSD over the standard McSplit.

3.4.3 McSplit2S

A more elaborate version, which generalizes both McSplitSD and McSplitSO, is also proposed by Trimble. This approach does not swap before starting the algorithm. Each time it selects the label class, it decides whether to branch on the left side of the label class or on the right one based on the size of each side of the label class itself.

To put it in simpler terms, given a label class $L = (L_G, L_H)$, if $|L_G| <= |L_H|$, it executes the standard McSplit algorithm, chooses a vertex v from L_G and then pairs all possible w inside L_G. If $|L_G| > |L_H|$, instead, it first chooses w from L_H and then tries to map it against all possible v vertices inside L_G.

While this approach can adapt dynamically at each branching, we will not use it since it would imply a substantial slowdown in our implementation of McSplitDAL.

3.5 Additional heuristics

This section will introduce the main heuristics we considered as possible substitutes to the Node Degree adopted by McCreesh in its original McSplit. The main reason for this choice is that using the degree of a vertex does not always lead to the best solution, hence the necessity of finding a good heuristic for classifying all the vertices of a given graph.

We combined all the following heuristics with our implementation of McSplitDAL, and we analyze their performance in section 3.6, where we will present our results.

3.5.1 PageRank

PageRank (PR) is an algorithm developed by Google which calculates the probability of reaching a page inside a network through a finite number of random clicks but was also used as a sorting algorithm for ranking search results. PageRank usually works on directed unweighted graphs, but it can be easily extended to undirected ones considering them as directed graphs with two edges between each node pair.

Algorithm 2 shows our PageRank, strongly inspired by a public version[1]. In our pseudocode, we use the notation $adj(G)$ to refer to the indices of the adjacency matrix of graph G. The Damping Factor (DF), initialized in line 1, represented a person's probability of stopping clicking random links. We chose to set $DF = 0.85$, as recommended in the original paper by Brin et al. ([10]). We also set the acceptable error ϵ at an arbitrary value such that it was precise enough not to overflow any 32-bit integer since we wanted to map ranks to integer numbers.

[1] https://github.com/purtroppo/PageRank

PageRank is a Markov chain. After having calculated the outgoing links, it computes a stochastic matrix. This matrix represents the quantity of information flowing through adjacent edges. It then transposes it since PageRank works on incoming links (and we calculated it on outgoing ones). The algorithm's core goes from line 26 to line 39. It starts with zeroed ranks and computes them at each iteration by adjusting the previous results, approximating the clicking probability, and discounting them by the DF. At line 36, we update the error and, at line 38, we update the result vector p. The algorithm iterates until convergence, stopping when the error is smaller than the precision set.

3.5.2 Closeness Centrality

The Closeness Centrality (CC) of a vertex evaluates its centrality inside a given graph and was first introduced by Bavelas et al. ([11]). This indicator can be calculated as the reciprocal of the sum of the lengths of all the shortest paths. It is worth noticing that a higher score implies a higher centrality because a central vertex has the smallest sum of shortest distances, hence having a higher closeness coefficient. Equation 3.5 formalizes the formula for computing this score.

$$CC(v_i) \quad = \quad \frac{1}{\sum_{v_j \in G} d(v_i, v_j)} \tag{3.5}$$

3.5.3 Local Clustering Coefficient

The Local Clustering Coefficient (LCC) was introduced by Watts et al. ([12]), and it measures the closeness of the neighborhood of a given vertex to be a complete graph (every node has a connection with all the others).

This measure can be calculated by counting, for each vertex, its neighbors (the degree of the node) and then counting the number of edges that connects the neighbors. Finally, the ratio between the neighborhood's connections over the total possible links between all the adjacent vertices to the starting one will output a score between 0 and 1, indicating the LCC for the given vertex.

We can extend this approach to undirected graphs by counting each edge twice (since it is present in both directions). Equation 3.6 formalizes it.

$$LCC(v) \quad = \quad 2 * \frac{|e_{jk} : v_j, v_k \in N_v, e_{jk} \in E|}{|N_v| * (|N_v| - 1)} \tag{3.6}$$

3.5.4 Betweenness Centrality

Betweenness Centrality (BC) is a relevant index in graph theory, first introduced by Freeman ([13]). This measure considers how many shortest paths pass through

Algorithm 2 Our version of the popular PageRank algorithm, implemented on an adjacency matrix representing the graph G

```
1: DF ← 0.85
2: ϵ ← 0.00001
3:
4: function StochasticGraph(G, out_links)
5:     G_s ← [0.0] * |G|
6:     for all x, y ∈ adj(G) do
7:         if out_link[x] = 0 then
8:             G_s[x, y] ← 1.0/|G|
9:         else
10:            G_s[x, y] ← G[x, y]/out_link[x]
11:        end if
12:    end for
13:    return G_s
14: end function
15:
16: function PageRank(G)
17:    out_links ← OutLinksForEachNode(G)
18:    G_s ← StochasticGraph(G, out_links)
19:    G_t ← TransposeMatrix(G_s)
20:    result ← ∅ * |G|
21:    p ← ∅
22:    for all x, y ∈ adj(G_t) do
23:        push(G_t[x, y]/|G|)
24:    end for
25:    error ← 1.0
26:    while error > ϵ do
27:        result ← ∅ * |G|
28:        for all x, y ∈ adj(G_t) do
29:            result[x] ← result[x] + G_t[x, y] * p[y]
30:        end for
31:        for all rank ∈ result do
32:            rank ← rank * DF + (1.0 − DF)/|G|
33:        end for
34:        error ← 0.0
35:        for all rank, prev ∈ zip(results, p) do
36:            error ← error + abs(rank − prev)
37:        end for
38:        p = result
39:    end while
40:    return result
41: end function
```

a given vertex v among all the shortest paths between any pair $(v, w) \in G$. It is very useful in a wide range of applications because it expresses the quantity of information passing through a given vertex inside a network, which is of uttermost importance in fields like telecommunications.

It is noteworthy to notice that even if it involves the shortest distances, it is different from Closeness Centrality since a *central* vertex, according to the betweenness criterion must be included in the shortest paths, not having small shortest paths leading to other vertices (simply being an endpoint). The implementation we used follows the formalization expressed by the following Equation 3.7:

$$BC(v) \quad = \quad \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{3.7}$$

where σ_{st} is the number of shortest paths from s to t and $\sigma_{st}(v)$ is the number of shortest paths from s to t passing through v.

3.5.5 Katz Centrality

The last heuristic we will take into account is called Katz Centrality (KC), introduced by Katz ([14]) in the 50s. This heuristic also measures the degree of centrality of a vertex, but instead of the shortest paths, it uses the number of walks between two vertices. It counts the number of adjacent vertices and all other vertices in the graph which can reach the given vertex through its neighbors. It introduces a penalty factor α for distant neighbors.

A formalization of Katz Centrality is Equation 3.8:

$$KC(v) \quad = \quad \sum_{k=1}^{\inf} \sum_{j=1}^{n} \alpha^k (A^k)_{vj} \tag{3.8}$$

where matrix A^k indicates wheter two vertices are connected in k walks.

However, due to the high cost of calculating this measure, we used a simplified version that considers the shortest walks only and not all the possible ones for each vertex.

3.6 Experiments

This section is organized as follows: first, we briefly introduce our experimental setup. After that, we analyze our implementation of McSplitDAL, showing the performances of the different versions up to the final one. Then, we will include the previously introduced heuristics into our McSplitDAL, showing the results of these additional changes. Finally, we will compare our best version of McSplitDAL against the classical McSplit and McSplitLL, which should be the most performant right after McSplitDAL, according to the recent literature.

3.6.1 Experimental setup

We executed all our tests on a workstation with an Intel Core i9-10900KF CPU and 64 GBytes of DDR4 RAM. We developed our algorithms in C++ and compiled them with GCC v9.4. McSplit and McSplitLL were taken directly from the authors' public implementations.

We ran our tests on two public datasets which differ in the size of the test instances: we called them *small* and *big*, respectively. The *small* dataset ([15]) contains graphs with slightly less than 100 nodes on average. We built it by choosing one pair of graphs for each category of the whole dataset, up to a total of 400 test instances. The *big* dataset ([16]) contains 733 graphs with a size ranging from 103 to 6474 vertices, and we built a dataset of 366 pairs from it, discarding one extra unpaired graph. We also defined a subset of this dataset called *big finetuning* made of 122 test instances, one-third of the whole dataset, which we extensively used for our tests presented in the following chapters. A more detailed analysis of these datasets will follow in Chapter 6.

Our goal is to evaluate the ability to find suitable solutions in a limited time frame since it is the closest application to real-world use cases. We fixed the timeout to 60 seconds since McSplit experimentally achieves a good solution on the first recursion tree in less than a second, spending most of the remaining time trying to improve it by searching in other branches, which does not always happen.

3.6.2 McSplitDAL

Our McSplitDAL follows all the ideas of Liu et al. ([8]), already introduced in Section 3.3, but we improved them by making some choices we validated on experimental results. We run all the tests presented in this section on *small* dataset.

Our first bare implementation had the following characteristics: it kept rewards in a vector for G and a matrix for H, sharing them between the RL policy and the DAL policy, switching them according to the threshold of iterations without improvement. The main differences between DAL and RL policies were:

- RL did not take into account the number of label classes generated in the reward
- RL did not decay, differently from DAL

We also kept the LUM introduced in McSplitLL since it provides a speedup with no additional costs.

The first improvement to this version was the introduction of McSplitSD. According to Trimble ([9]), it should mainly provide benefits and is not too complex to be integrated, in contrast to the generalized McSplit2S.

Figure 3.1 compares the two approaches. The plot is a normalized rolling average and shows the size of the solutions of each method, normalized to the McSplitDAL results, averaged over windows of 50 consecutive tests. We also ordered the tests by the size of the average solution found to see the behavior on tests with growing complexity. We can see that McSplitDAL+SD stays above McSplitDAL most of the time, implying once again an overall improvement. Moreover, the normalized rolling average shows that McSplitDAL+SD can gain a solid margin over McSplitDAL when it wins. The positive spike at the center of the graph confirms our hypothesis since it is way higher in absolute value than the negative spike at the right, which indicates an average win of McSplitDAL.

Therefore, given the results, we can assume that the SD swap policy addition is beneficial.

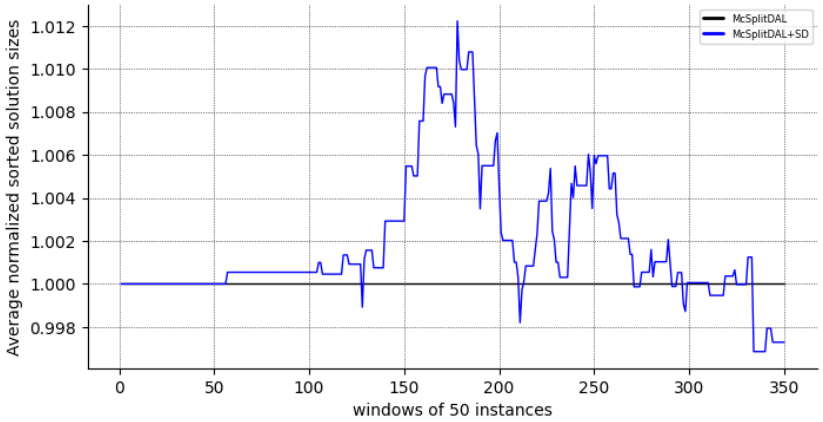

Figure 3.1: Rolling average comparison of McSplitDAL+SD

The second improvement we made to our McSplitDAL was the separation of the rewards. As we described, our first implementation kept DAL and RL rewards together, changing only the update method after a policy switch or activating/deactivating the decay according to the policy. This choice has two main drawbacks: RL policy should not depend on LSM (H should have a vector of rewards that never decays), and the two policies' rewards should not overlap. Our solution was to store a vector and a matrix to separate them, updating both according to their policy, but only the current policy's rewards were used to select the vertices at a given point of the algorithm. We named this approach with isolated rewards as *McSplitDAL+SD iso*.

Following the comparison strategy used for showing the benefits of McSplitDAL+SD, we plot again a normalized rolling average between McSplitDAL+SD and its *iso* version. Figure 3.2 confirms that the separation of the rewards led to significant gains in terms of solution size, with this new version dominating over

the previous one on all tests.

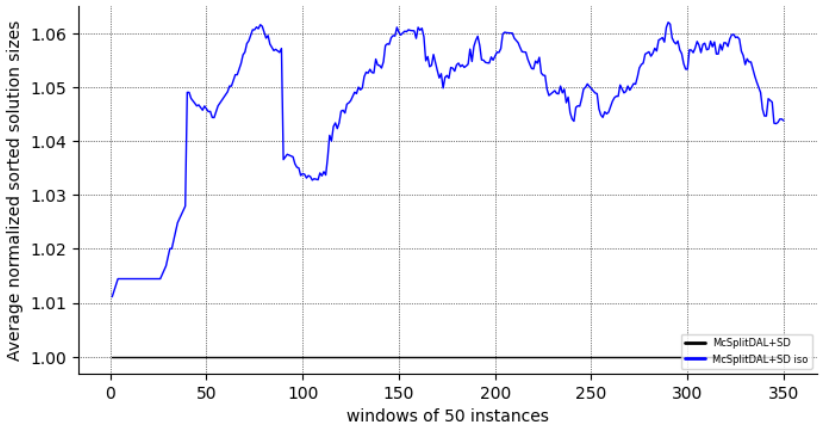

Figure 3.2: Rolling average comparison of McSplitDAL+SD iso

The last consideration on our McSplitDAL did not lead to an improvement, but it was worth exploring and regards the reward initialization. We tried to initialize the rewards of each vertex to its heuristic value instead of having rewards initialized to zero. The name of this variant is *McSplitDAL+SD iso_init*, and we show a comparison against the *McSplitDAL+SD iso* version on Figure 3.3, using the usual normalized line plot.

We can notice that neither of the two variants completely overpowers the other one, but the new one using the *init* strategy, seems to be highly unstable, and in the right part of the plot, it has a significant negative spike.

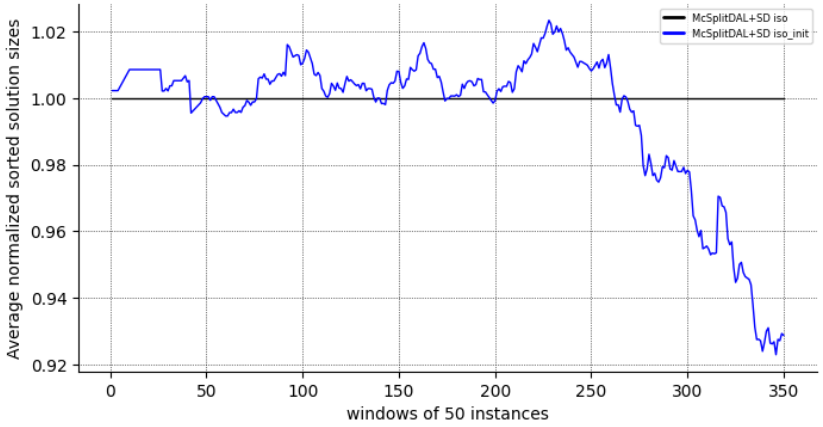

Figure 3.3: Rolling average comparison of McSplitDAL+SD iso_init

We can use the heatmap computing the performance gain of each method to

visualize which is, *on average*, the best solution, even if we know for sure that the one without initialization is the most stable.

The metric used to calculate the performance gain is an average over the normalized difference of results. Given two methods, we compute the difference between their results. Then, we normalize it over the average result of the two methods. Finally, we average all these normalized differences, obtaining the performance gain of method 1 over method 2.

Figure 3.4 shows the described comparison and allows us to state that the best solution overall is *McSplitDAL+SD iso*.

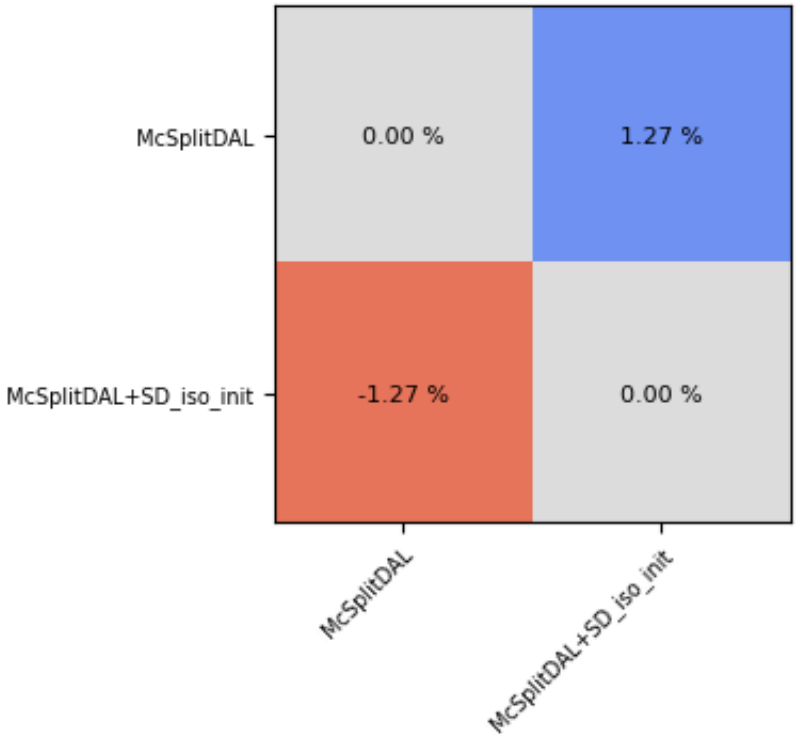

Figure 3.4: Average performance gain heatmap of McSplitDAL+SD iso_init

From now on, we will call the best version simply as *McSplitDAL*, since it will be the basic block we will use for our subsequent tests.

3.6.3 Heuristics

This section evaluates how our McSplitDAL's performances change if we substitute the Node Degree heuristic with one of our heuristics.

We ran tests both on the *small* and on the *big* datasets to assess if some heuristics

behave better or are consistent across different graph sizes. The evaluated heuristics are six, leading to the following McSplitDAL variants:

- McSplitDAL, using Node Degree heuristic
- McSplitDAL+PR, using PageRank heuristic
- McSplitDAL+BC, using Betweenness Centrality heuristic
- McSplitDAL+LCC, using Local Clustering Coefficient heuristic
- McSplitDAL+CC, using Closeness Centrality heuristic
- McSplitDAL+KC*[2], using Katz Centrality heuristic

Figure 3.5 and Figure 3.6 show the line plot with the rolling average of these variants on *small* and *big* datasets respectively, using McSplitDAL as a baseline for the comparison. We extended the rolling window to 100 to allow a cleaner visualization of the patterns, since there are many methods compared.

We can notice that on the *small* dataset there are some good methods like McSplitDAL+KC* and some which performs worse, like McSplitDAL+CC. However, on the *big* dataset, we see a clear distinction between variants which perform worse than the baseline and variants which consistently perform better.

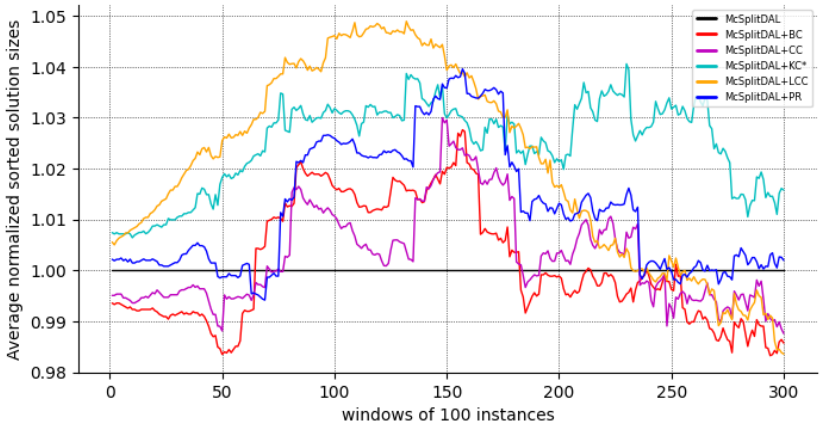

Figure 3.5: Rolling average comparison of McSplitDAL with different heuristics on *small* dataset

[2]We use a simplified Katz Centrality which considers shortest paths only

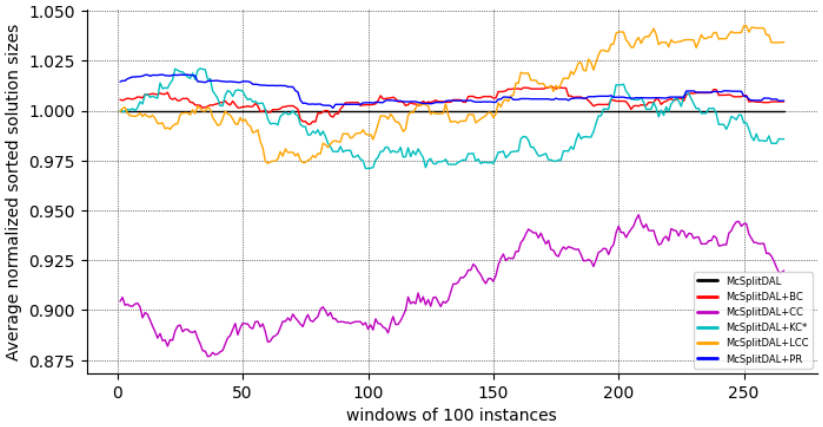

Figure 3.6: Rolling average comparison of McSplitDAL with different heuristics on *big* dataset

As we did previously, we also show head-to-head comparisons using heatmaps.

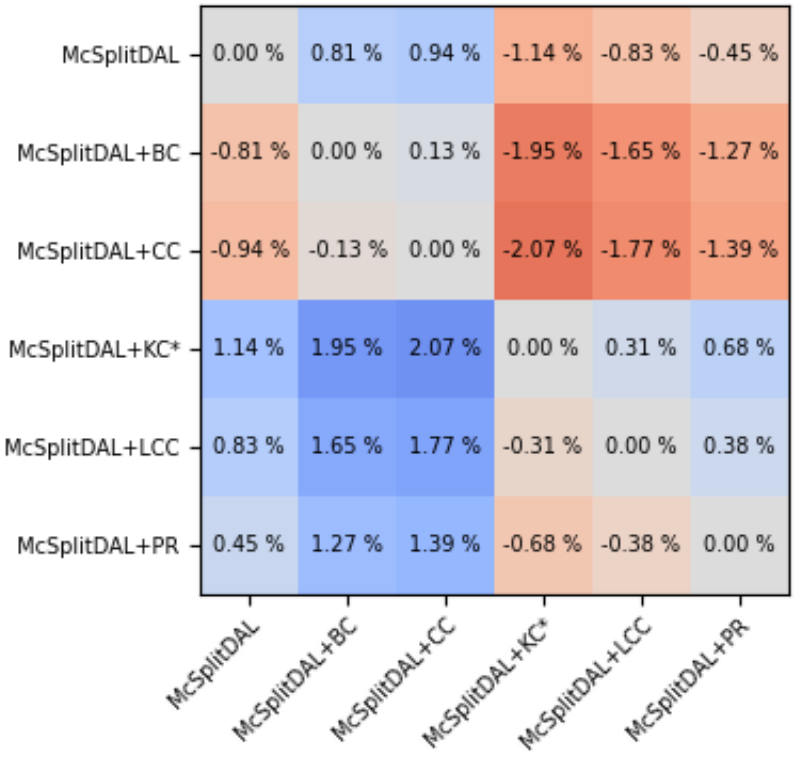

Figure 3.7: Average performance gain heatmap of McSplitDAL with different heuristics on *small* dataset

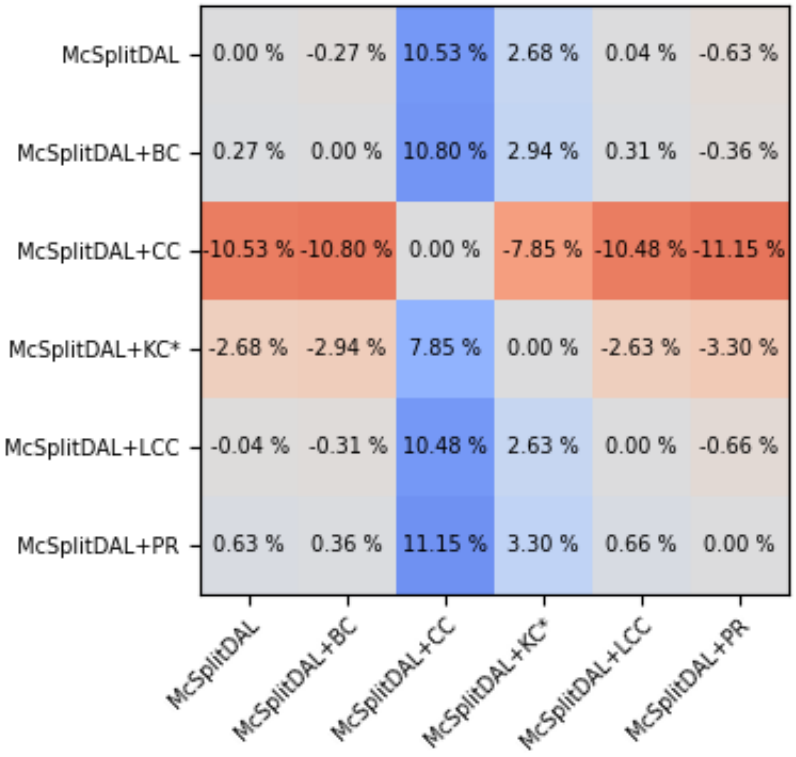

Figure 3.8: Average performance gain heatmap of McSplitDAL with different heuristics on *big* dataset

Figure 3.7 and Figure 3.8 quantify how much each method outperforms another method, allowing us to estimate which heuristics provide an improvement over the Node Degree.

Table 3.1 recaps the average performance gain of each proposed method over the McSplitDAL baseline using Node Degree.

Method	Gain on *small*	Gain on *big*
McSplitDAL+BC	-0.82%	**0.27%**
McSplitDAL+CC	-0.94%	-10.53%
McSplitDAL+KC*	**1.14%**	-2.68%
McSplitDAL+LCC	**0.83%**	-0.04%
McSplitDAL+PR	**0.45%**	**0.63%**

Table 3.1: Average performance gain over McSplitDAL with Node Degree using different heuristics

The results suggest there is no absolute winner, but we can make some considerations.

PR seems to be the most stable heuristic since it performs well on both datasets. Also, the LCC heuristic seems to be beneficial on the *small* dataset, while it provides an almost identical performance on the *big* dataset, as to the Node Degree. BC seems to be a valid heuristic for big datasets, while it does not behave well on smaller ones. KC* has the opposite behavior of BC, working well with simple graphs but degrading its performance as complexity grows. Finally, CC does not perform well on both datasets, so it is our worst heuristic.

3.6.4 Results

Our main goal was to outperform current state-of-the-art methods, so we need to compare them.

We choose our McSplitDAL+PR as our candidate since it seems to be the most stable among the explored combinations, and we compare it against McSplitLL, which is the best method after McSplitDAL according to the literature, and the original McSplit.

As usual, we employ a line plot to evaluate the average performance and then quantify it using a heatmap. We ran these tests on the *small* dataset since our method is stable in both datasets and potentially less performing on the *small* one.

Figure 3.9 show the line plot using the usual rolling average metric and highlights the performance boost provided by our method, which outperforms both McSplitLL and the original McSplit.

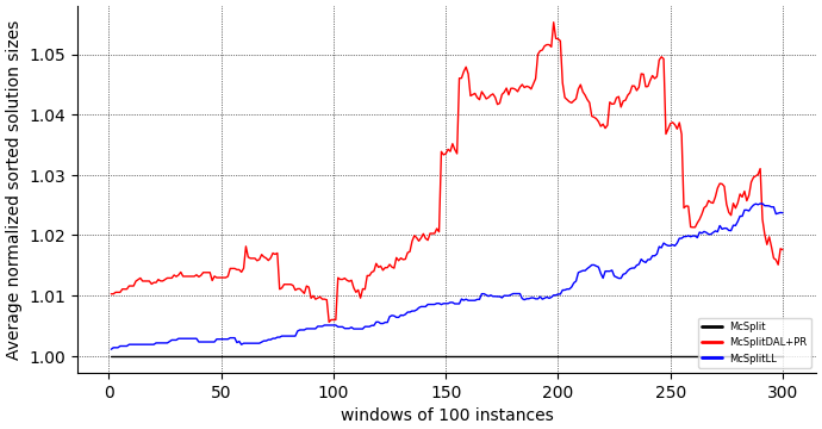

Figure 3.9: Rolling average comparison of McSplitDAL+PR, McSplitLL and original McSplit on *small* dataset

To better quantify how much is the performance boost, we refer to Figure 3.10,

which provides the percentage of gain comparing each method head-to-head. Our McSplitDAL+PR provides a positive performance boost respecting the other two strategies, as expected from the line plot, hence establishing the new state-of-the-art for MCIS exact solvers using the McSplit algorithm on a single thread.

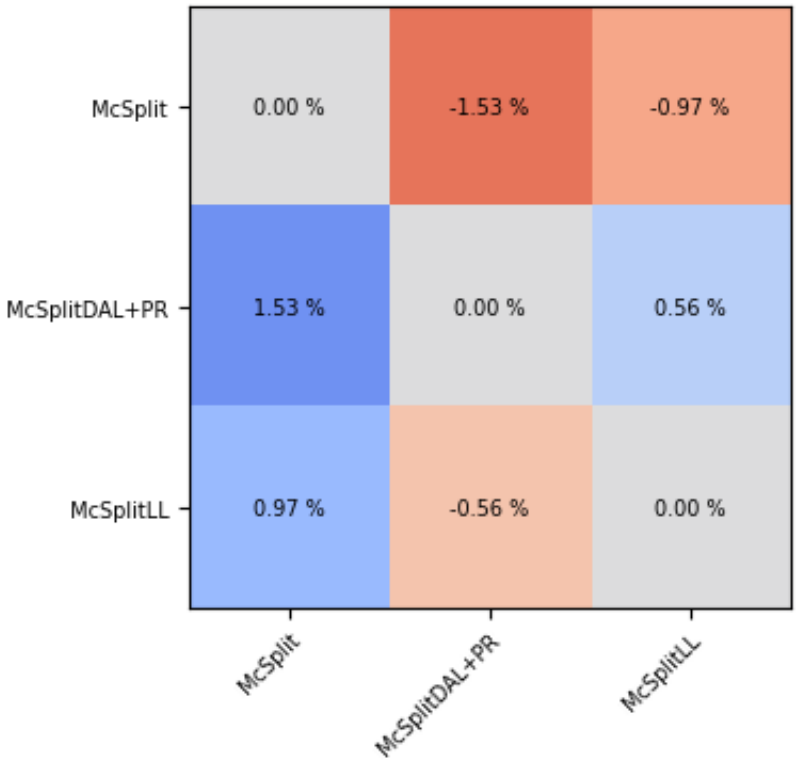

Figure 3.10: Average performance gain heatmap of McSplitDAL+PR, McSplitLL and original McSplit on *small* dataset

Chapter 4

McSplit and Parallelism

This chapter will focus on enhancing McSplit algorithms by exploiting parallelization. Specifically, we will introduce two alternative algorithms: McSplit Multi-Branch (MB) and McSplit Branch-Sharing (BS).

The former is an extension of the parallel version developed by McCreesh and Trimble, while the latter is a new approach written by us.

After a theoretical introduction aimed at explaining these two approaches, we will provide our results in the last section of this chapter.

4.1 McSplit Multi-Branch

McSplit Multi-Branch is a parallel version of the original McSplit, which can explore the search space much faster than the single-thread version, gaining a boost thanks to the many threads working on the task simultaneously.

This approach follows the standard recursive implementation of McSplit, but it divides the load among different threads by exploiting a task queue up to a maximum depth. When the algorithm starts, the main thread pushes a task selecting the first vertex v from the left graph G and iterating over the vertices w of the right graph H having the same label class as v, we will define this first level having $depth = 0$. Instead of recurring and continuing solving, this thread pushes the context for this level of depth as a task to the global queue. At this point, the waiting threads start from the context passed, either picking a different w or backtracking and choosing another v. All these threads are working at $depth = 1$, and instead of recurring themselves, they will each push a task to the queue.

This strategy continues until the depth becomes bigger or equal to a given threshold. Then, instead of pushing to the task queue, threads will solve the task with a depth higher than the threshold without sharing the work with other parallel workers.

The main advantage of this load balancing is the speed and the low memory impact since it should work as possible on shared variables and is limited to a low depth (equal to 4, in our case). However, the small value of the depth threshold is also a disadvantage since, on big graphs, it could narrow down the search since only a small part will be visited with a shared approach, while, after the sharing ends, only one thread could keep a potentially good search tree, without any possibility of parallelizing it.

We extended this approach to adapt to the heuristics introduced in Chapter 3. It is also worth mentioning that, since this is merely an extension of an already existing implementation, we did not use any of the ideas introduced by recent variants of McSplit exploiting Reinforcement Learning, namely McSplitRL, McSplitLL, and McSplitDAL, which we introduced in Chapter 3.

4.2 McSplit Branch-Sharing

McSplit Branch-Sharing is a brand new approach whose aim is to overcome the limitation of McSplit Multi-Branch due to the unbalancing introduced by the depth threshold but also to be flexible enough to accommodate new McSplit variants like McSplitDAL.

This algorithm still follows all McSplit's original ideas about label classes and search pruning, but it has a different structure since it is iterative instead of the classical recursive one. This choice simplifies splitting the load among threads. The core unit of work is now the step which is the context at a given depth of the search tree. It can be a *V-step* if it selects the current v from the left graph G, or a *W-step* if it pairs a w vertex with the v selected on its parent *V-step*.

Another difference between this parallel implementation and McSplit Multi-Branch is that this one incorporates both heuristics and also recent variants of McSplit, namely McSplitSD and McSplitDAL, leveraging both the power of Reinforcement Learning and the effortless benefits of the SD swap policy.

The pseudocode is presented in Algorithm 3 and has two core states: global and local. The former state is a stack that can be accessed in mutual exclusion by any thread to get a step to work with. The latter acts like a buffer containing the steps fetched from the global stack. Whenever a worker has a non-empty local stack, it pops the first step and processes it like a V-step or a W-step. A V-step becomes a W-step after the first possible w is selected, and a W-step becomes a V-step either including the selected pair (v, w) or not (hence being a backtrack step). Each newly created step goes into the local stack, and eventually, two possible actions can be taken, leading to a different variant of McSplit Branch-Sharing.

The standard McSplit Branch-Sharing version uses a local stack until it reaches a given size, passed as an option to the algorithm. When the stack becomes full, the

Algorithm 3 Pseudocode of McSplit Branch-Sharing

```
1: Best ← {}
2: Steps_global ← stack(S_0)
3:
4: function McsBranchSharing(G, H)
5:     Steps_local ← stack()
6:
7:     while time < timeout do
8:         Pop Step sg from Steps_global under mutual exclusion
9:         Steps_local.push(sg)
10:        while 0 < Steps_local.size() < block_size do
11:            s ← Steps_local.getHead()
12:            Restore context from s
13:            if s is a StepV then
14:                if |Best_current| > |Best| then
15:                    Best ← Best_current
16:                end if
17:                Bound ← ComputeBound(G, H, |Best_current|)
18:                if Bound < |Best| then
19:                    return
20:                end if
21:                LC ← SelectLabelClass(G, H)
22:                v ← SelectVertexV(G, β)
23:                G' ← G' \ {v}
24:                Convert s to StepW
25:                Steps_local.push(s)
26:            else
27:                w ← SelectVertexW(H, β)
28:                Best'_current ← Best_current ∪ {v : w}
29:                H' ← H \ {w}
30:                UpdateLabelClasses(G', H', v, w)
31:                s_next = StepV(context)
32:                Steps_local.push(s_next)
33:                if LC_right = ∅ then
34:                    Steps_local.pop()
35:                end if
36:            end if
37:        end while
38:
39:        if Steps_local.size() > 0 then
40:            Steps_global.push(Steps_local.all())
41:        end if
42:    end while
43: end function
```

local steps are pushed to the global stack. This action will enable sleeping threads to share a part of the load, leveraging parallelism and allowing them to work in parallel on promising search trees, which is impossible under McSplit Multi-Branch assumptions.

The second variant operates *until pruning*. At the start of the algorithm, only one thread works. In the previous approach, as soon as it generates enough steps, it splits the load by pushing it to the global stack and sharing it with other threads. It is beneficial for parallelism but does not exploit the first search tree, which is usually explored in a few seconds and leads to a big solution. Therefore, sharing the workload too early can cause a broader exploration that cannot benefit from the pruning which would occur if the first search tree was fully explored.

With this in mind, instead of using a size limit on the local stack from the start, the first thread will keep working locally until a pruning occurs. When pruning happens, it sets the size limit and pushes the steps to the global stack. This approach combines the advantages of the promising first search tree of MCIS and the smart workload sharing of McSplit Branch-Sharing.

Finally, we also allow removing the features of our McSplitDAL (rewards, LUM, hybrid branching, and swap policy) to understand if they are beneficial with multiple threads.

4.3 Experiments

This section aims to evaluate the performance of these methods on the *big* dataset presented in Chapter 3, Section 3.6.1. We decided to focus on this dataset because the parallel benefits should be more evident on larger graphs, and it is easier to finetune parameters when the gap is observable.

The presentation of the results follows the model of the preceding chapter: first, we analyze each method separately, then we compare them.

We perform finetuning of each proposed algorithm on the *big finetuning* dataset, which is one-third of the original dataset in size. This decision was due to the high number of tests required for a complete comparison over a wide range of possible configurations.

4.3.1 Finetuning McSplit Multi-Branch

McSplit MB's performance changes by varying the number of threads or using a different heuristic. It is worth noticing that this method does not exploit Reinforcement Learning, since it does not use McSplitDAL but uses plain McSplit, therefore changing the heuristic could have a bigger impact on the tests.

We split the analysis of results into two parts: finetuning the number of threads and assessing heuristics' influence. First, we observe the effectiveness of parallelism

by varying the number of threads keeping the heuristic fixed. After that, we select the best number of threads and keep it fixed, varying the heuristic.

Finetuning McSplit Multi-Branch: threads

The tests on the degree of parallelism will use the PageRank heuristic since we showed in Chapter 3 its effectiveness and stability across different classes of test instances.

We apply tests with 1, 4, 8, 16, 24, and 32 threads. We did not exceed 32 since our CPU has 20 virtual cores, and the benefits, if any, would surely be less evident.

Figure 4.1 shows the line plot with the rolling average of McSplit MB + PageRank with a variable number of threads. We use the single-thread version as a baseline for comparison, and we can notice how the performance increases with increasing thread size.

As we expected, the gain in terms of results when the number of threads exceeds the number of virtual cores of the CPU is much lower, even if still present.

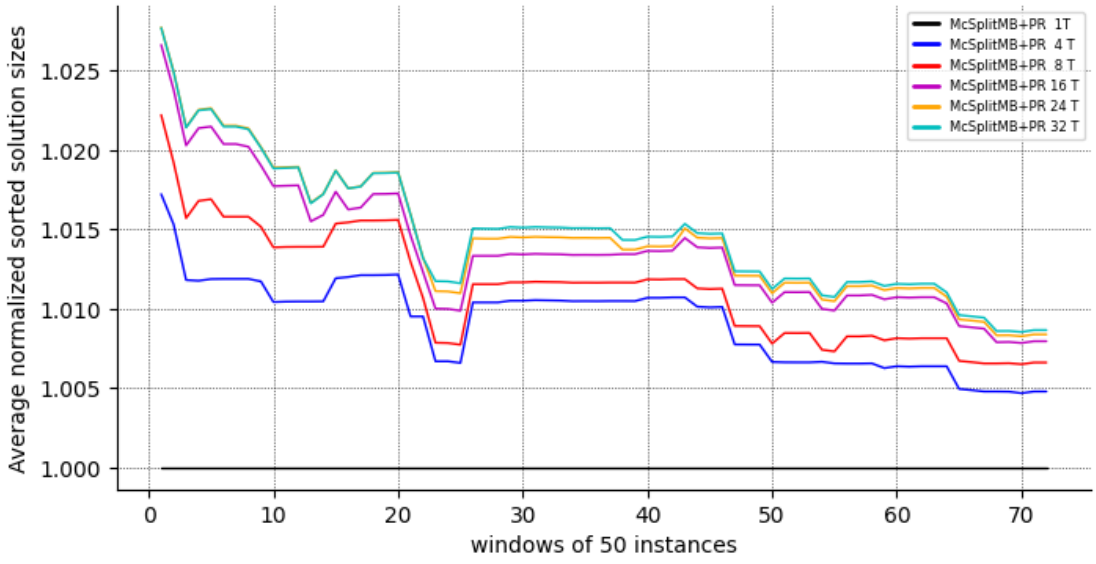

Figure 4.1: Rolling average comparison of McSplitMB+PR with different threads on *big finetuning* dataset

We can observe the gain as a percentage through the heatmap in Figure 4.2. The lower triangular matrix shows that increasing the number of threads leads to better performance over the same method. Moreover, we can see that the boost is almost negligible when we switch to 24 and 32 threads, while it is more evident up to 16 threads.

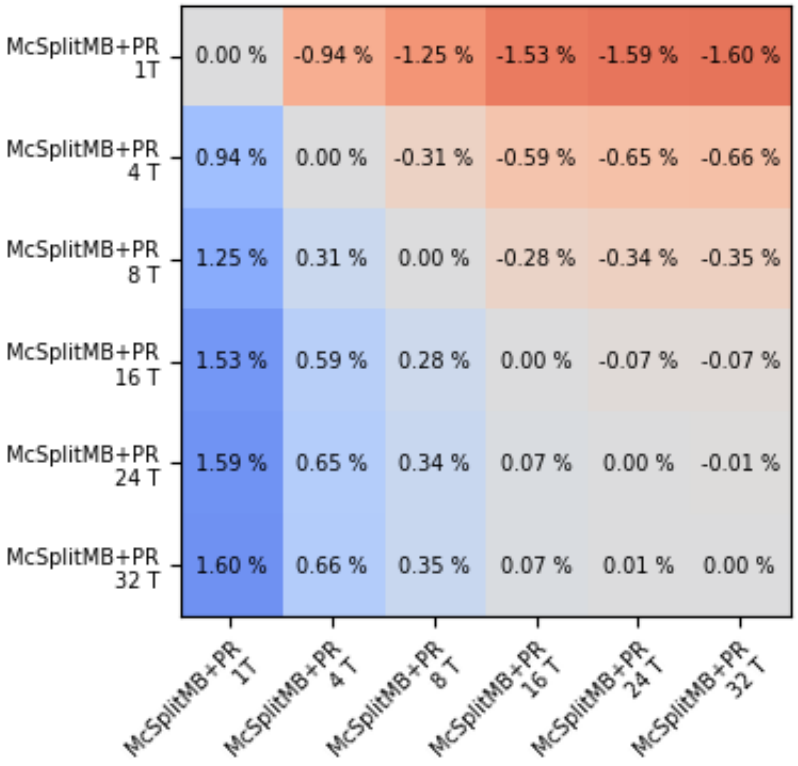

Figure 4.2: Average performance gain heatmap of McSplitMB+PR with different threads on *big finetuning* dataset

We can conclude that our best configuration uses 32 threads, being the most performant of all, even if by a slight margin.

Finetuning McSplit Multi-Branch: heuristics

After selecting the best number of threads to parallelize McSplitMB, which we set equal to 32, we need to validate how the performance varies when we use a different heuristic. We will use the heuristics introduced in Chapter 3, comparing them with the original McSplitMB version, which uses the Node Degree.

Figure 4.3 compares the behavior of the heuristics through a line plot. Using a different heuristic provides benefits, except for Closeness Centrality, which is again the worst. Our modified Katz Centrality has the best performance, but the remaining heuristics can outperform the Node Degree too.

As for comparing the effect of a different number of threads, we show the heatmap of gains in Figure 4.4, which confirms the previous claims on the best and worst methods.

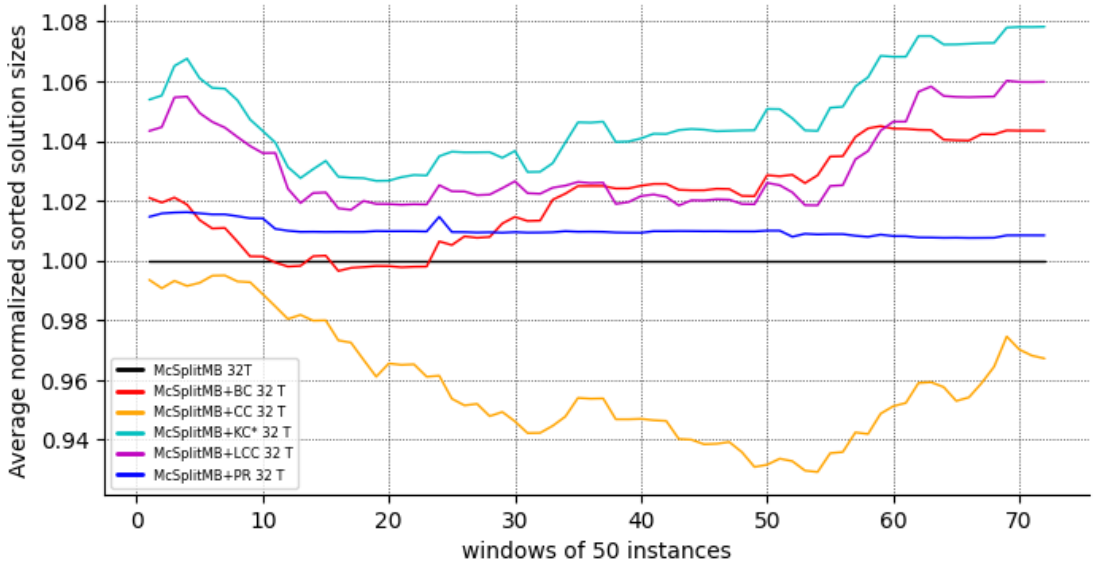

Figure 4.3: Rolling average comparison of McSplitMB 32T with different heuristics on *big finetuning* dataset

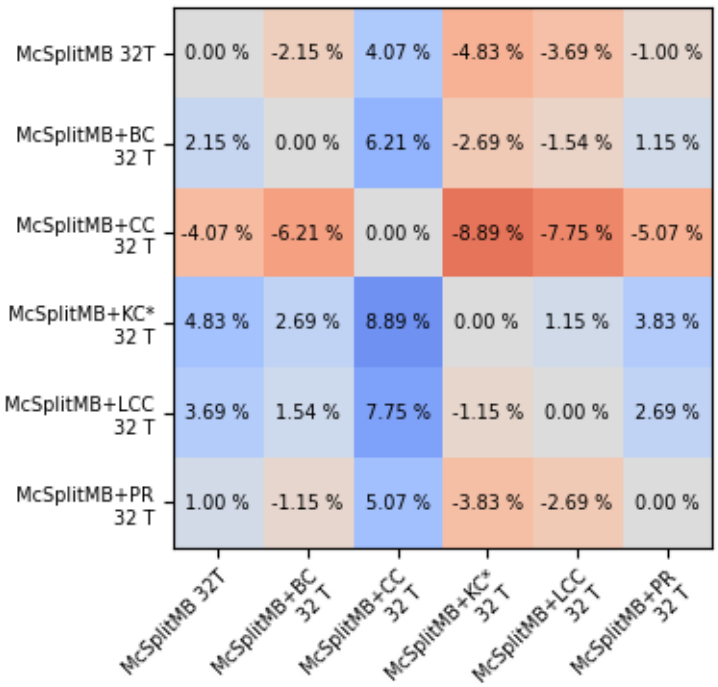

Figure 4.4: Average performance gain heatmap of McSplitMB 32T with different heuristics on *big finetuning* dataset

4.3.2 Finetuning McSplit Branch-Sharing

Our McSplitBS differs from McSplitMB in load balancing management, introducing a new parameter called block size. The block size is the maximum size the local thread queue can reach before sharing the created steps with other threads through the global stack, and we have to finetune it too. Therefore, we will need to evaluate many variants of McSplitBS based on: block sizes, number of threads, heuristics, usage of until pruning, and usage of McSplitDAL.

Finetuning McSplit Branch-Sharing: threads and block size

To compare the behavior of our McSplitBS with different thread numbers, we also need to evaluate reasonable block sizes. However, it is not convenient to directly compare the numbers of threads with a range of block sizes since it would be impossible to interpret plots due to the many versions possible.

Therefore, we fix a heuristic, PageRank, and 24 or 32 threads. Then we compare how the block size affects the performance. Finally, we select the most promising block size and run the tests to validate a fixed block size over different numbers of threads. As we did with McSplitMB, we use the *big finetuning* dataset to perform our tests.

We analyze first how block size affects our McSplitBS run on 24 threads. Figure 4.5 shows the line plot over these block sizes: 8, 16, 32, 64, 128, 256, and 512. We use $B = 8$ as a baseline to normalize the results of the other combinations, and we can see that there is a gain up to a block size, then the performance settles to a slightly lower value.

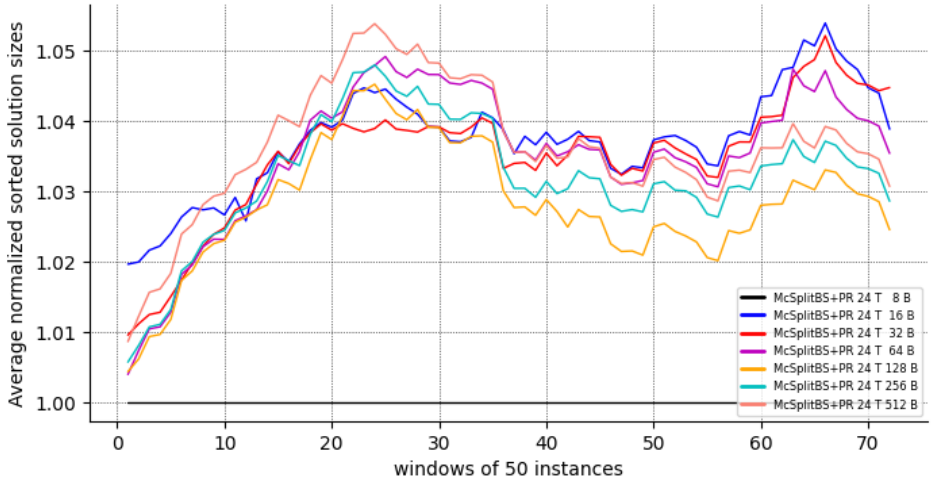

Figure 4.5: Rolling average comparison of McSplitBS+PR 24T with different block sizes on *big finetuning* dataset

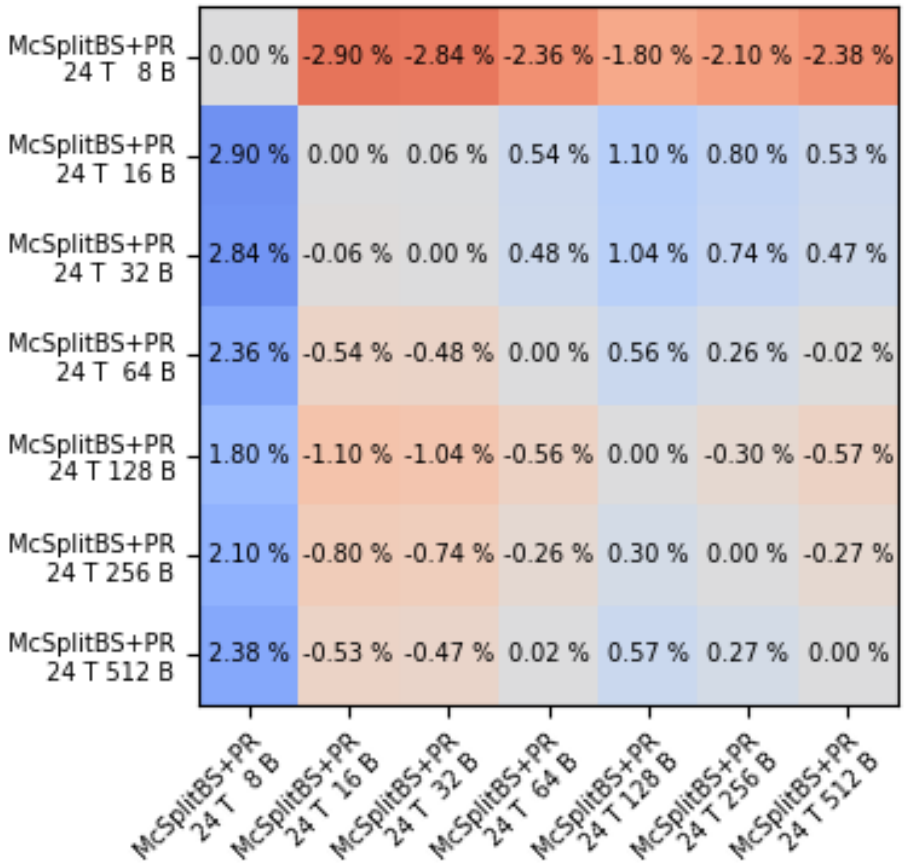

Figure 4.6: Average performance gain heatmap of McSplitBS+PR 24T with different block sizes on *big finetuning* dataset

The best block sizes are 16 and 32, and we can verify it by looking at the heatmap of gains in Figure 4.6, which confirms our hypothesis.

We ran the same experiments on McSplitBS using 32 threads and obtained similar results, with $B = 32$ being the best combination by a neat margin. To verify our claims, we can once again refer to the line plot and the heatmap of gains reported in Figure 4.7 and Figure 4.8, respectively.

These results suggest that a block size that is not too big or too small provides better results. We can explain it by observing that a small block size forces threads to push to the global stack more often, performing a broader search over different trees. Larger block size has the opposite behavior since it allows less work sharing and a deeper exploration of the current search tree. A middle point of the two extrema benefits of both strategies.

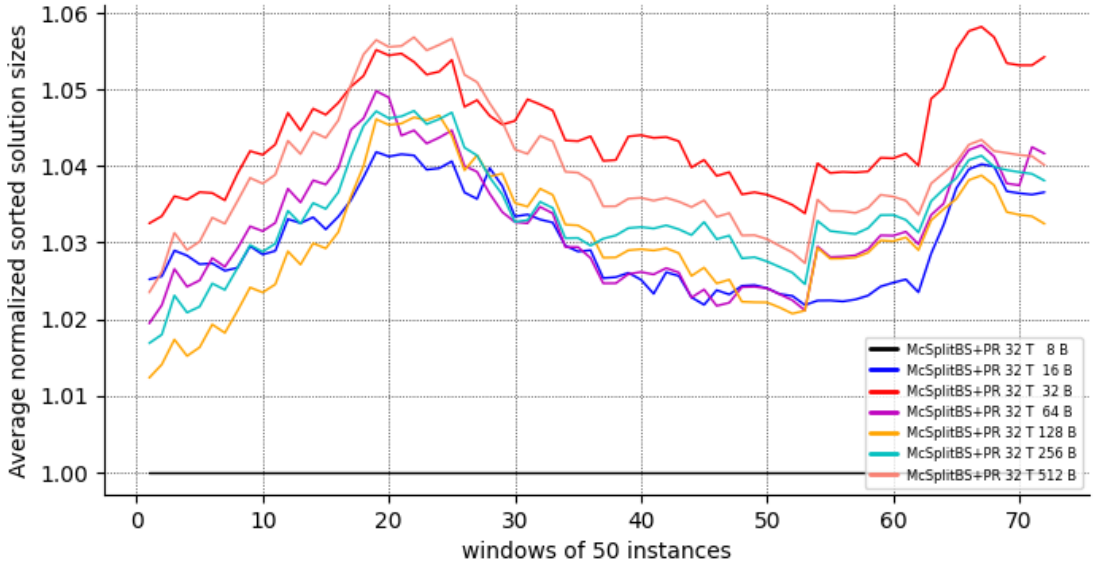

Figure 4.7: Rolling average comparison of McSplitBS+PR 32T with different block sizes on *big finetuning* dataset

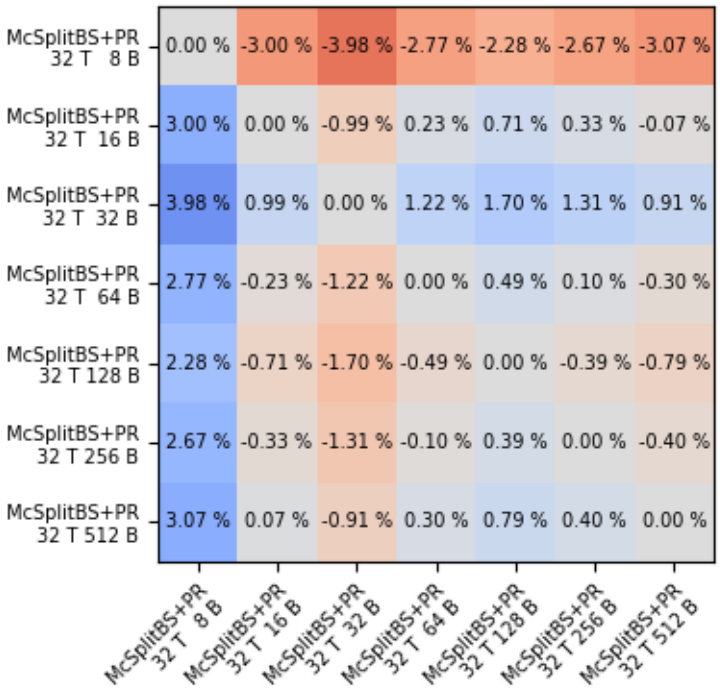

Figure 4.8: Average performance gain heatmap of McSplitBS+PR 32T with different block sizes on *big finetuning* dataset

To conclude our analysis, we select $B = 32$ as block size, since it is the best on 32 threads and is almost on par with $B = 16$ on 24 threads, and we run the tests using 4, 8, 16, 24, and 32 threads.

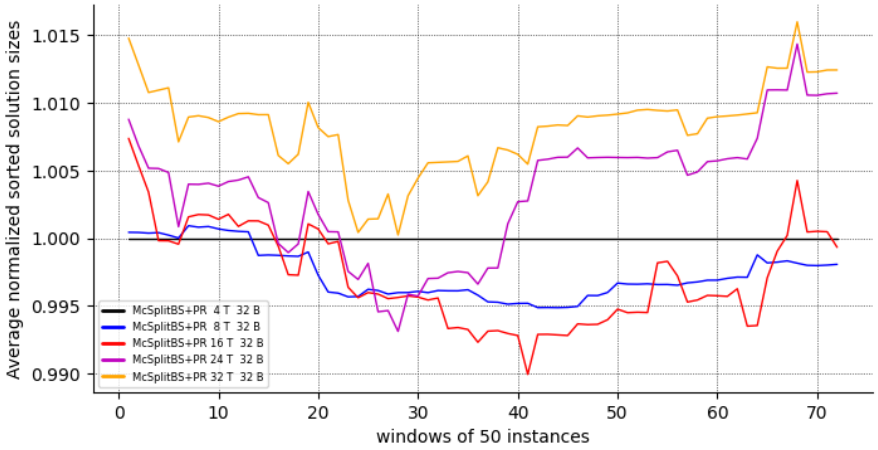

Figure 4.9: Rolling average comparison of McSplitBS+PR 32B with different number of threads on *big finetuning* dataset

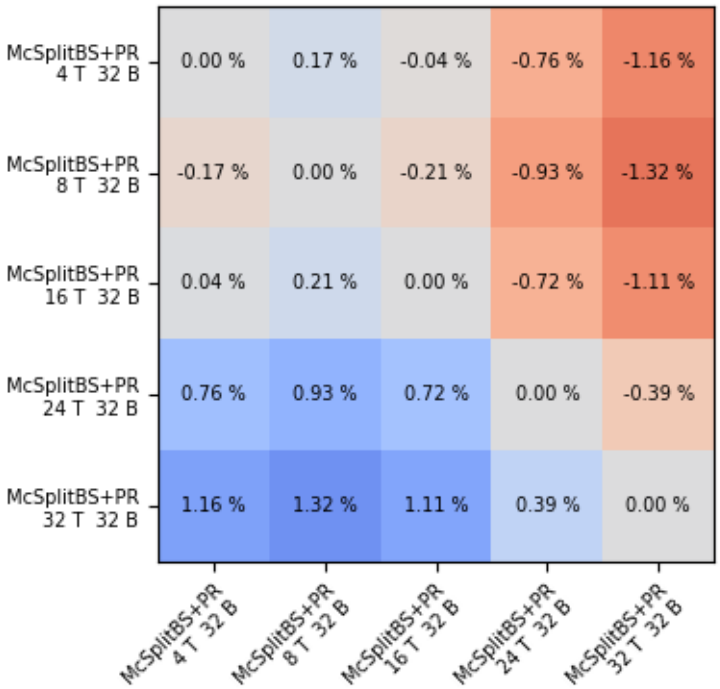

Figure 4.10: Average performance gain heatmap of McSplitBS+PR 32B with different number of threads on *big finetuning* dataset

Figure 4.9 reports the line plot, which validates the effectiveness of using a higher number of threads, using the variant with 4 threads as a reference to normalize the results. Figure 4.10 also gives us the numerical percentage of gain of each method against the others, with the 32 threads variant able to be the best overall. It is worth noticing that the variant with 8 threads seems less performant than the one with 4 threads: this is mainly due to the overhead introduced by McSplitBS for managing parallelization and its weak point, the execution speed. Whenever the number of threads is too low, the overhead reduces performance: a higher parallelization is necessary to overcome this issue.

We can conclude that the most promising configuration is McSplitBS with 32 threads and block size = 32, which will be our baseline in the next finetuning steps.

Finetuning McSplit Branch-Sharing: heuristics

As we did with McSplitMB, we want to understand how the heuristic influence our performance so we run our tests using McSplitBS with 32 threads and a block size equal to 32, while varying different heuristics. We report the rolling average line plot on Figure 4.11 with the Node Degree heuristic as a baseline. We can see that the best heuristics seem to be: Betweenness Centrality, PageRank, and Node Degree. The other three heuristics score lower in terms of performance, and this can be verified on the heatmap shown in Figure 4.12.

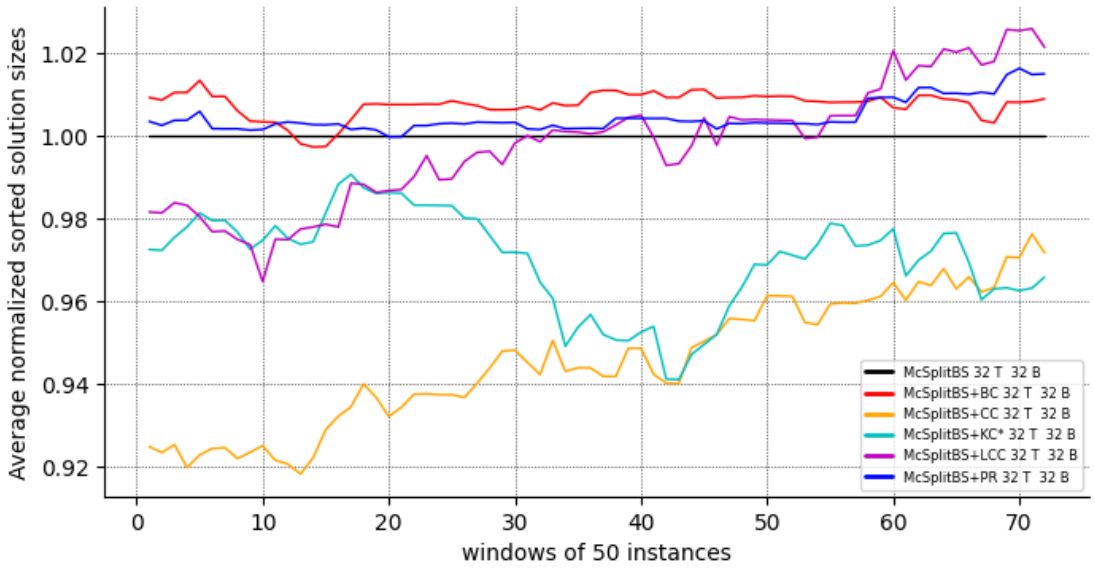

Figure 4.11: Rolling average comparison of McSplitBS 32T 32B with different heuristics on *big finetuning* dataset

We can notice that the results differ from McSplitMB but are similar to our McSplitDAL presented in Chapter 3. The reason could be the likeness of the two approaches: McSplitBS initially explores the first search tree like McSplitDAL, since it does not parallelize the work until the local stack is full. On the other hand, McSplitMB initially splits the work as much as possible and after that starts to work separately in each thread. This difference could be the reason behind the obtained results.

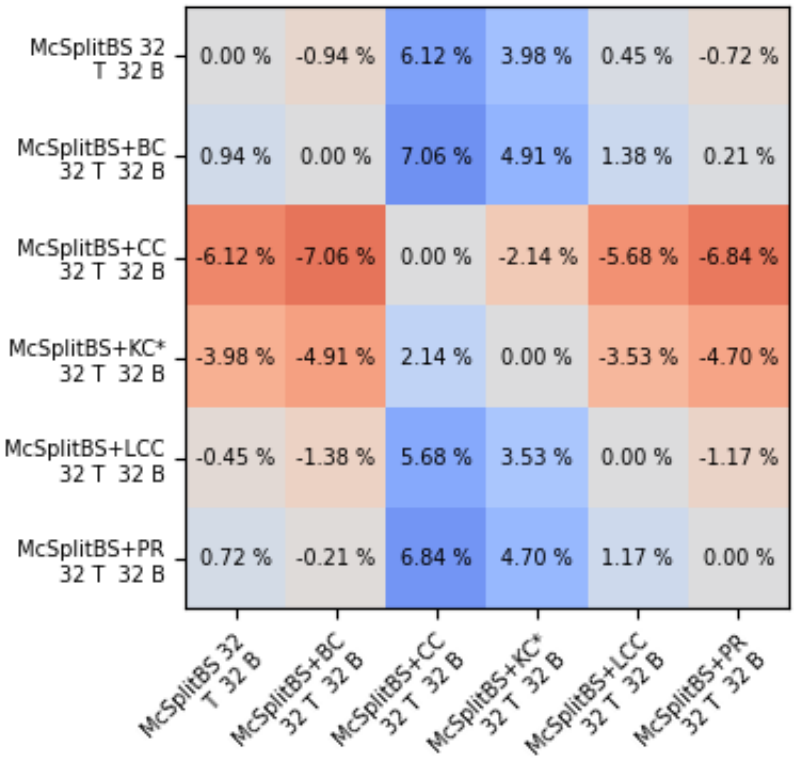

Figure 4.12: Average performance gain heatmap of McSplitBS 32T 32B with different heuristics on *big finetuning* dataset

Finetuning McSplit Branch-Sharing: until pruning

One of the possible variants of McSplitBS does not limit the block size at the start of the algorithm. As we explained, it is possible to set a flag called *until pruning*, which does not allow sharing steps with other threads until the first reaches a pruning condition.

To test whether this is beneficial or not, we employ our McSplitBS 32T 32B with the top three heuristics, and we compare the variants having the until pruning

flag active against the ones without the flag. Figure 4.13 and Figure 4.14 reports the line plot and the heatmap of performance gains, respectively.

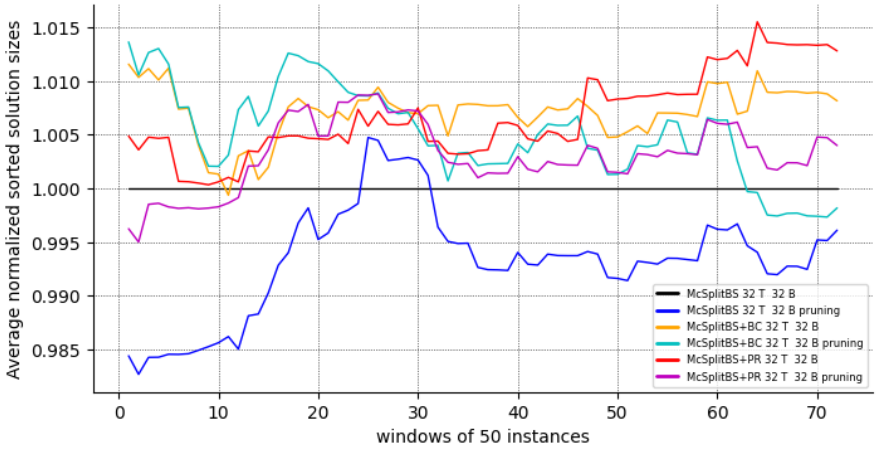

Figure 4.13: Rolling average comparison of McSplitBS 32T 32B until pruning on *big finetuning* dataset

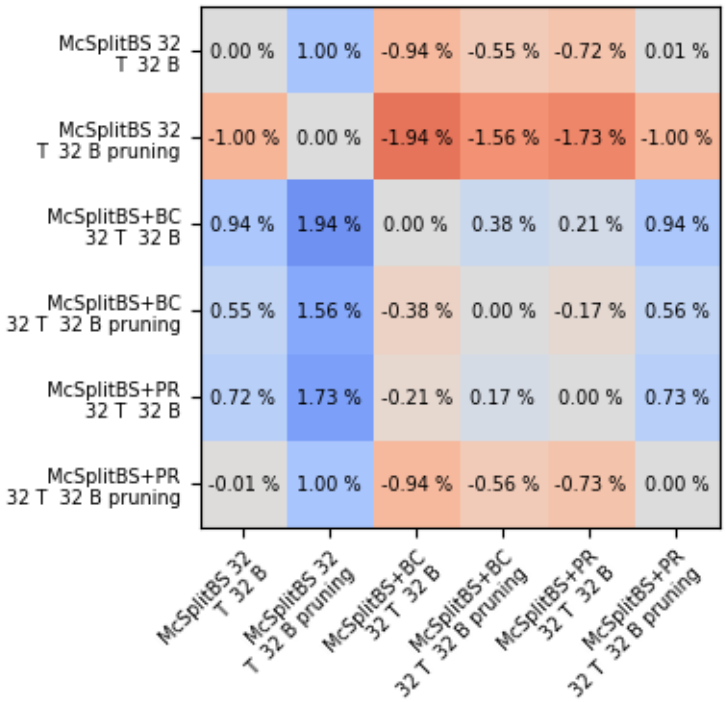

Figure 4.14: Average performance gain heatmap of McSplitBS 32T 32B until pruning on *big finetuning* dataset

We can see from the heatmap of performance gains that none of the methods using *until pruning* can outperform its standard counterpart, and the line plot shows similar behavior too. Observing these results, we can conclude that it is not worth exchanging the initial parallelism for searching for a good solution on the first recursion tree because it is detrimental to performance: the time spent to reach the first pruning is better spent by sharing the workload. Therefore, we will not use this feature in the next tests.

Finetuning McSplit Branch-Sharing: without McSplitDAL

Finally, we want to see if McSplitDAL is truly effective when the number of threads increases, and we want to test our top three variants with different heuristics with McSplitDAL and without it by using the original McSplit without rewards, LUM, swap policy, et cetera. This test is important because the high degree of parallelism could make the rewards unreliable since we often change search three.

We show our results using Figure 4.15 and Figure 4.16, which allow us to visualize the correctness of our hypothesis. Indeed, the *no dal* variants seem to be more performant and to obtain better results than their counterpart using McSplitDAL's advanced techniques.

We can thus conclude that our best variants of McSplitBS run on 32 threads with a block size equal to 32, and do not use DAL. Moreover, the best heuristics with this method are Node Degree, Betweenness Centrality, and PageRank.

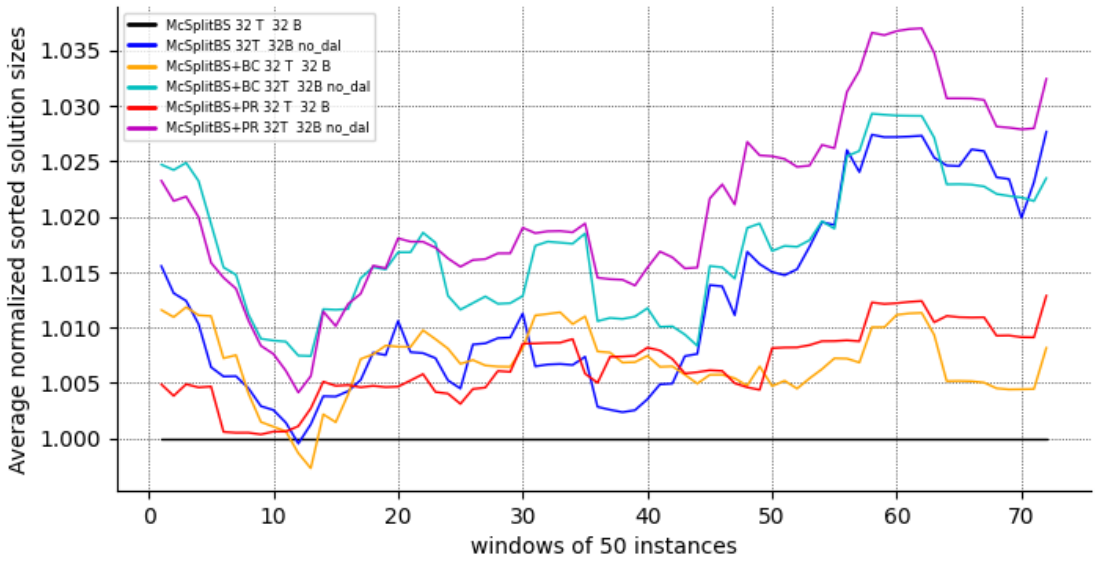

Figure 4.15: Rolling average comparison of McSplitBS 32T 32B without McSplitDAL on *big finetuning* dataset

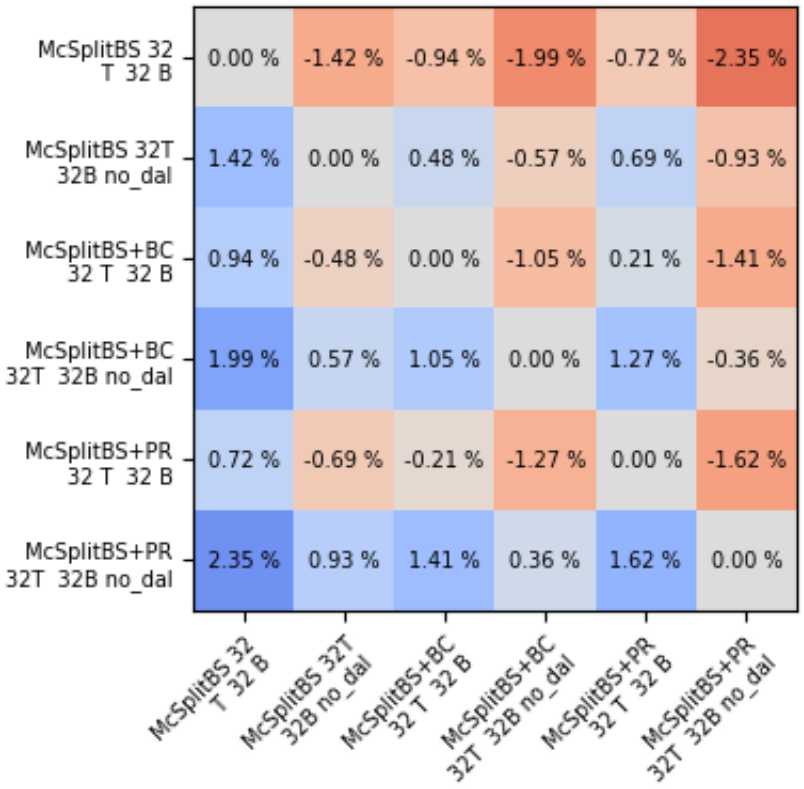

Figure 4.16: Average performance gain heatmap of McSplitBS 32T 32B without McSplitDAL on *big finetuning* dataset

4.3.3 Results

This last section compares the best versions of McSplitBS and McSplitMB with the original parallel version, which is publicly available and coincides with our McSplitMB using 32 threads and the Node Degree heuristic, to assess the effectiveness of our methods.

We show our results using the line plot having the original version as a baseline to normalize the results (Figure 4.17) and the heatmap of performance gains (Figure 4.18) to quantify the improvement of each method and evaluate the best ones.

Looking at the plots, we see that our parallel versions outperform the basic McSplitMB using Node Degree heuristic, and the best ones are McSplitMB+BC 32T, McSplitMB+LCC 32T and McSplitBS+PR 32T 32B no DAL. It is interesting to see that our McSplitBS+PR and McSplitBS+BC win against McSplitMB+BC, proving the potential of our new algorithm even if it can be largely improved, especially in terms of execution speed.

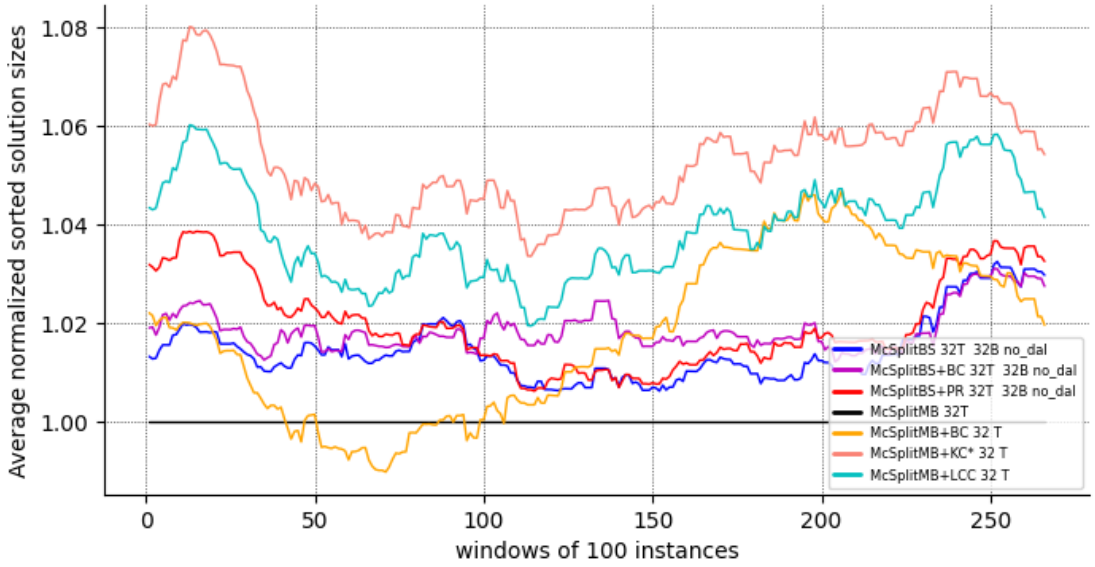

Figure 4.17: Rolling average comparison of McSplitMB and McSplitBS on *big* dataset

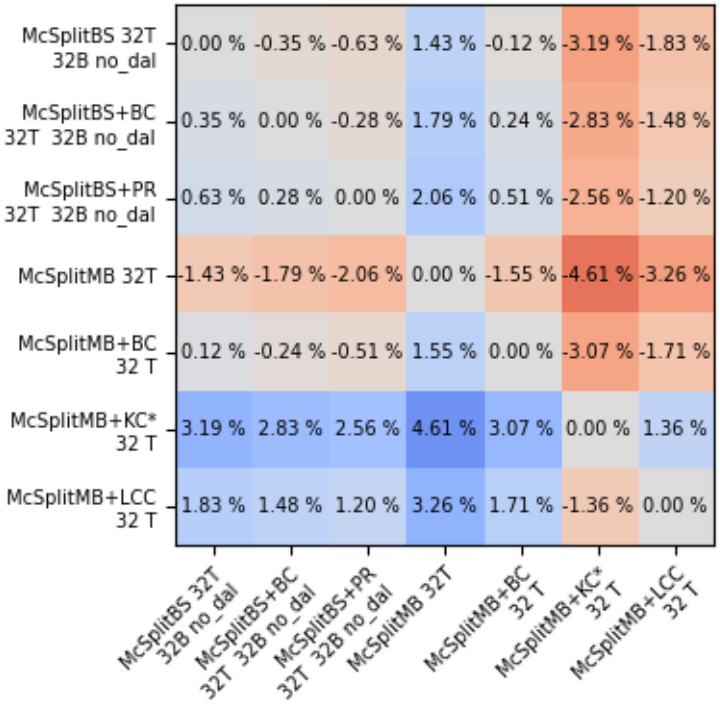

Figure 4.18: Average performance gain heatmap of McSplitMB and McSplitBS on *big* dataset

Chapter 5

McSplit and Graph Neural Networks

We will now shift our attention to a different approach based on Deep Learning. Specifically, this chapter will contain three possible solutions using Graph Neural Networks: GLSearch, McSplitGNN, and McSplitDiffGNN. GLSearch is a famous method in the literature, known for successfully introducing the usage of GNNs for solving the MCS problem. McSplitGNN and McSplitDiffGNN, instead, are new approaches presented here for the first time, aimed at building a simple but efficient GNN architecture for solving the MCS problem. After a theoretical explanation of both approaches, we show our experiments and analyze the results of our new techniques. However, due to the limitations of GLSearch explained in Section 5.1, we will not present comparisons involving GLSearch.

5.1 GLSearch

GLSearch is a model based on Graph Neural Networks specifically trained for solving the *connected* version of the MCIS problem. It has been introduced by Bai et al. ([2]) and is the first known approach in literature aiming at solving this task using Deep Learning.

In the following sections, we show an overview of the algorithm, its architecture, and the training procedure devised by the authors. Finally, we analyze its limitations.

5.1.1 Overview

McSplit and its variants based on Reinforcement Learning suffer limitations which can lead them to find small solutions.

The Node Degree heuristic is not adaptive to real-world graphs, hence providing an inefficient ranking of nodes.

Theoretically, the newer version based on Reinforcement Learning can overcome this limitation by introducing a strategy based on rewards, which should be able to adapt without using a static heuristic. However, this approach exposes an additional issue: the absence of a training stage. The missing training stage makes the RL policy restarts from scratch each time the environment (the input graphs) changes. However, the biggest problem is the starting zero-init of the rewards. In the initial phase, having all rewards equal to zero leads to breaking ties using the Node Degree heuristic, degenerating to the McSplit original heuristic. Even if this happens only in the initial stages of the algorithm, it could lead to exploring a huge search space not inspectable in a reasonable amount of time.

GLSearch wants to remove this limitation and proposes a trainable model able to adapt to the problem without learning a strategy from scratch, and that does not degenerate into the original McSplit's heuristic.

5.1.2 Architecture

The core idea of GLSearch is creating a trainable Reinforcement Learning approach. The authors were able to realize it by exploiting a combination of Deep Q-Learning and Graph Neural Networks to choose the best action inside a given context.

Using a suitable training approach (described in the next section), the model learns to select the best pair at each state by choosing the one having the highest reward inside the DQN built for the current label class. For each possible couple, the algorithm calculates the reward of choosing it as the next action leveraging the power of a Graph Convolutional Network with Attention (GAT). The GAT combines node-level embeddings, local neighborhood embeddings, and graph-level embeddings to project interactions into a compact representation.

This architecture is thus able to process also large graphs, jointly maximizing the expected reward and the MCS size. This approach is different from the standard McSplit approach since the latter usually aims at reducing the search tree size rather than directly maximizing the solution.

5.1.3 Training

Due to the intrinsic difficulties of the MCS task, it is not straightforward to train a model able to solve efficiently this task autonomously. Since the action space is ample, GLSearch authors divided the training procedure into three phases: pre-training, imitation learning, and autonomous training.

The first phase is a supervised approach run on small graphs, which can be fully explored (removing the pruning) by McSplit. In this phase, the DQN is trained for

each action using an MSE loss between the true reward (equal to the largest MCS from that state to a terminal one) and the predicted reward of the model.

The second phase focuses on graph pairs that cannot be fully explored anymore. Therefore, the loss is computed by using the McSplit heuristic as the true reward, finetuning the DQN predicted reward. Its name is *imitation phase* because McSplit acts as an expert agent to guide the GLSearch trajectory through the heuristic.

Finally, the last stage uses the *epsilon-greedy method* to choose with a probability ϵ whether to use the DQN prediction or to make a random choice. This approach is beneficial in the initial steps of this stage, where the approximation of the DQN may still be inaccurate, while it slowly becomes less necessary while the model learns to solve the task. The probability follows a decay schedule and settles to a low positive value, balancing the exploitation of the learned approach with the exploration of random sampling.

5.1.4 Limitations

The approach introduced by GLSearch has been proven beneficial on very large graphs by the authors. However, it has two main limitations: the task directly targets the connected variant of MCIS without options to use it to generate disconnected MCIS, and it is not possible to run the public implementation against different datasets.

We tried to modify the code to accommodate these two requirements but could not produce a working version. Moreover, we could not train the model from scratch, despite talking with the authors directly.

Due to these obstacles, we could only run the pre-trained models to solve the connected variant of the MCIS on the restricted dataset proposed by the authors and validate their results.

It would have been interesting to rewrite the whole implementation to make it more flexible and testable against different datasets to allow a direct comparison against the proposed methods inside this thesis. However, we could not accomplish such a task due to time constraints and will leave it as a future work.

5.2 McSplitGNN

The first of our two proposed architectures is McSplitGNN, which consists of two models: one chooses vertices from the left graph, and the other chooses from the right one.

The following sections propose two variants to this approach which uses the GNNs in opposite ways. The former exploits the model dynamically to select the vertices at each iteration. On the other hand, the latter uses the GNNs to statically sort the two graphs before running the algorithm.

5.2.1 Overview

This strategy follows the ideas of GLSearch: obtaining a trainable architecture able to run on graphs of varying sizes and structures while also finding valid solutions. However, we decided not to use Deep Q-Learning but to rely entirely on Graph Neural Networks to encode node features.

We build two identical models (one for each input graph) and then directly use them to get the predicted v and w while maintaining the McSplit algorithm.

Since we aim to work also on big graphs, we used a sequential version of McSplit adapted to this task.

5.2.2 Architecture

The architecture is made up of two twin models, both having the same base structure. The model takes a label class as input and outputs a vector of scalar scores for each vertex inside the label class. The algorithm selects the vertex with the highest score inside the label class.

The architecture consists of multiple layers: there are three Graph Convolutional Networks, each followed by a ReLU activation layer. The GCNs work on the node starting with one feature and ending with eight features per node (they are increased twice by a factor of 4 and then halved by the last GCN). After these convolutions, two Linear Neural Networks downscale the node features to 4 and then to 1. The first linear layer is followed by a ReLU activation, while the last one is followed by a sigmoid, which outputs a scalar score for each of the input nodes. The presented architecture is shown on Figure 5.1.

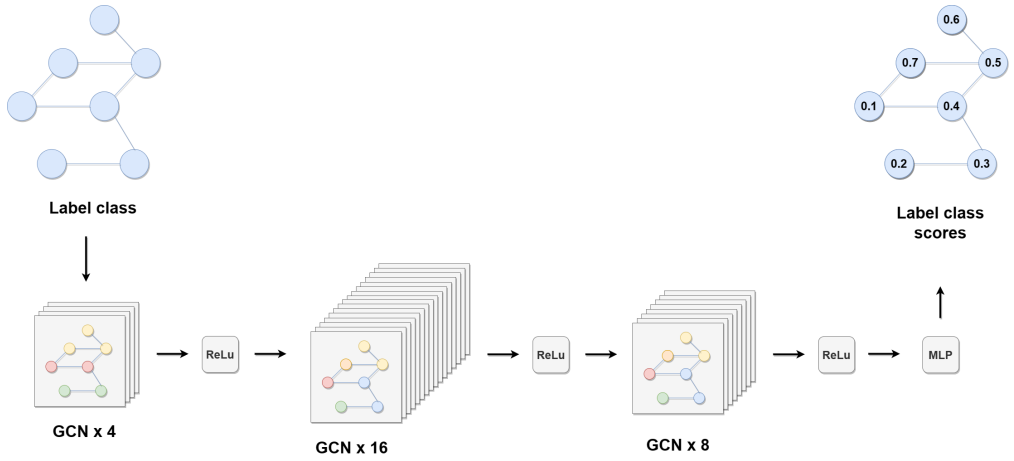

Figure 5.1: McSplitGNN model architecture

The model's scores can be used in two ways: as a static heuristic or as a dynamic one.

The static approach is similar to the one presented in Chapter 3, thus becoming an additional heuristic for the initial sorting step before running our RL-based McSplit version. This strategy has two advantages: it is fast and allows exploiting the GNN knowledge in the initial phase when the rewards are not stable yet.

On the other hand, the dynamic variant entirely substitutes the Reinforcement Learning approach for selecting the next vertex (either from the first or second graph). As we highlighted before, we have two twin models, so we can choose either to use both or only one of the two. In the latter case, the policy is hybrid since the RL framework is used only for one vertex selection, while the other uses the GNN-based model.

5.2.3 Training

The main goal of our model is to output scores such that it selects the best pair from a given label class. Therefore, we wrote a McSplit version adapted to export data during its execution. We decided to export, at each selection, the current label class and a score equal to the increment of the solution size obtained by choosing that vertex at the current step. Moreover, each vertex has two possible labels: 0 if it was not part of the final MCS or the score exported otherwise.

During the training phase, we load these data and calculate the vertices' scores for each label class. Then, an MSE loss is applied according to the real labels to make the predictions as similar as possible. The end goal is that the model outputs non-zero scores for the vertices which have to be part of the solution, where a higher score represents a higher chance of getting a bigger solution.

5.2.4 Limitations

Due to time constraints, we used an architecture with small node features inside the GCN layers, and we did not use more time-expensive GNN-based layers. Moreover, we run the training phase on a limited quantity of graph pairs (10). Therefore, the amount of extracted label classes was low, and the small variability of input graphs could hinder performance.

5.3 McSplitDiffGNN

McSplitDiffGNN is an architecture based on a single model, different from McSplit-GNN. This model only selects w vertices to match with the v vertex chosen by our RL-based McSplit presented in Chapter 3.

In the following sections, we analyze this technique, explain our architectural choices, and show its limitations.

5.3.1 Overview

This approach has the same goals as McSplitGNN but also exploits the power of Reinforcement Learning. Instead of giving a score to vertices, the model aims at encoding nodes such that vertices from the same label class have similar representations. Therefore, it first chooses a vertex from the left graph through an RL policy, then the most similar vertex from the right graph having the same label class is selected, exploiting the model.

We will present two versions of this model retaining the same architecture but trained on different datasets.

5.3.2 Architecture

The model accepts a label class as input and outputs a matrix containing one vector for each vertex inside the label class. These vectors encode each vertex over N features, and we chose $N = 64$.

We built the architecture using Graph Convolutional Networks and activation layers. We used four GCNs, each followed by a ReLU activation except for the last one, followed by a sigmoid. Each GCN increases the number of features of each vertex, which starts with one and grows up to 4, then 16, then 32, and finally 64 features. We decided to limit the number of features to balance results and time spent training. Figure 5.2 allows to visualize the architecture.

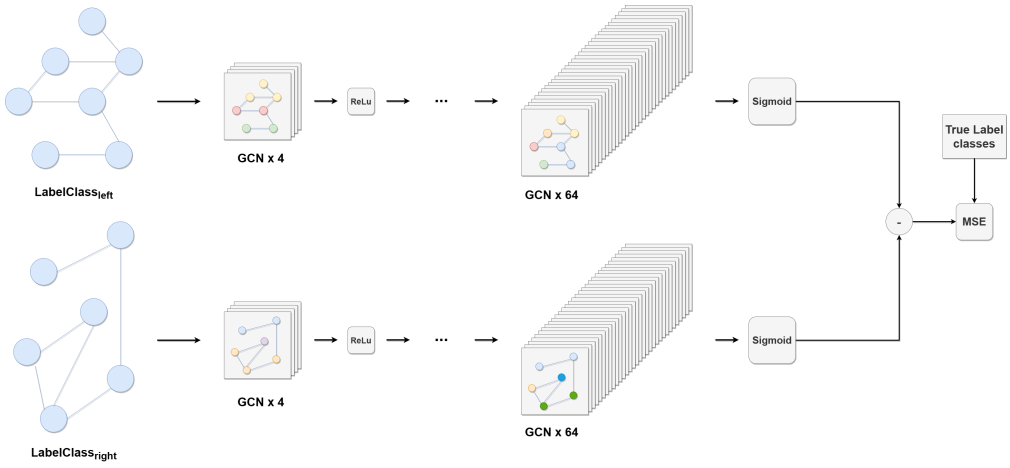

Figure 5.2: McSplitDiffGNN model architecture

5.3.3 Training

The trained model should output a similar representation for vertices sharing the same label class.

We used a similar dataset structure as in McSplitGNN: for each iteration of an ad-hoc McSplit version, we exported the label class selected and the pair of vertices inside it. Each pair of vertices had a label equal to 1 if it was present in the final solution, 0 otherwise.

During the training, we iterate over all the possible pairs of vertices and generate their embeddings using our model for each label class. Then, we balance embeddings such that we have an equal number of couples for each label. Finally, we compute the absolute difference between the embeddings from the left graph and the embeddings from the right one, and we compute the MSE Loss according to the following criterion: if a pair is inside the solution, the difference of embeddings must be a zero vector. Otherwise, the difference must be a vector with features equal to 0.5. This requirement is due to the final sigmoid that projects the codomain of the embeddings between 0 and 1. Therefore, requiring a difference equal to 1 would mean that the final embeddings should be infinities of opposite signs, hindering the optimization.

We also trained the model on a synthetic dataset created from an existing MCIS and added vertices and edges, generating two similar input graphs. We built this dataset to train the model on big pair of graphs whose resulting MCIS was known. These would not be possible otherwise due to the complexity of the tasks when the input graphs grow in size.

5.3.4 Limitations

This strategy has a potential drawback in the training dataset. While processing the label classes, we may find some vertices in multiple label classes of varying sizes. This repetition can be problematic if some vertices appear in small or big label classes since their embeddings may be too specific or "general" (many vertices would have similar embeddings). It could be beneficial to finetune a parameter to filter the size of the label classes so that the embeddings produced are a reasonable compromise between the two extrema. However, we could not perform a hyperparameter optimization due to time constraints and left it as future work.

5.4 Experiments

This section focuses on the results obtained by our GNN architectures. As in the previous chapter, we chose to run our experiments on the *big finetuning* dataset presented in Chapter 3, Section 3.6.1. This choice was motivated by the main goal

of McSplit-based GNN: being able to process larger graphs through a reusable and trainable model.

It is worth mentioning that these results are obtained by models trained for a limited amount of time and epochs due to time constraints, so they are not to be considered definitive. In the following subsections, we compare our model using a modified McSplit, which chooses vertices randomly, to assess that our models learn a proper relationship better than going by chance.

5.4.1 McSplitGNN's performance

McSplitGNN comes in 4 different flavors, which we can classify into two types: a static version which behaves as a new static heuristic, and a dynamic one made up of two twin GNNs that choose from one of the input graphs each. We can further split the latter version into three variants: using both GNNs, using only the left one on G, or using only the right one on H. All the GNNs underwent the same training: a dataset of 10 graph pairs for three epochs.

Figure 5.3 compare our methods with the random baseline using a line plot with a rolling average. Our methods outperform the random strategy by far, with the static variant leading the group in performance. The heatmap of performance gains (Figure 5.4) allows us to see these results from a quantitative point of view, confirming the effectiveness of our models.

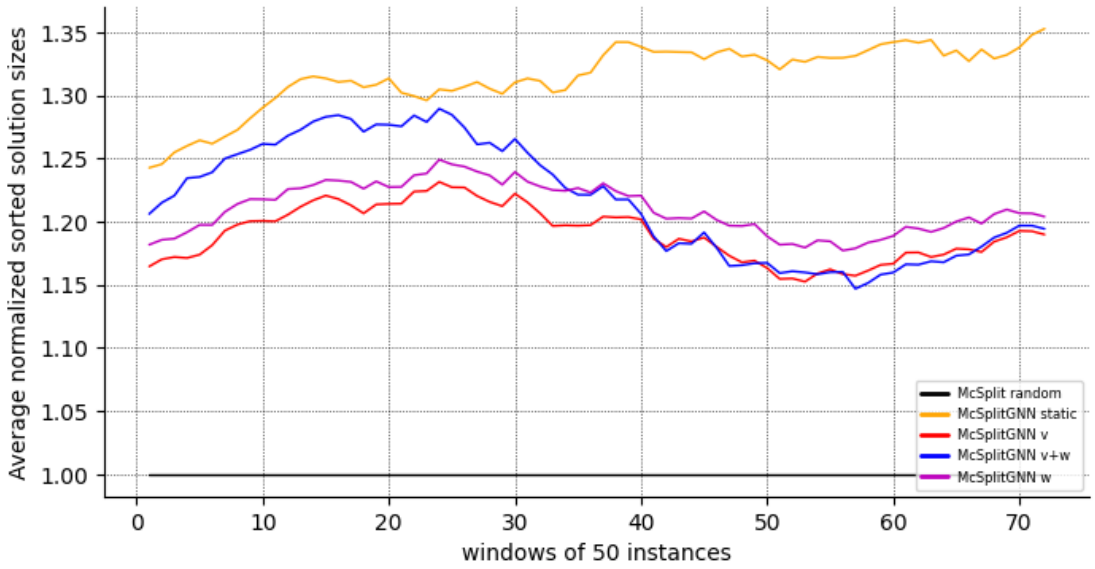

Figure 5.3: Rolling average comparison of McSplitGNN variants on *big finetuning* dataset

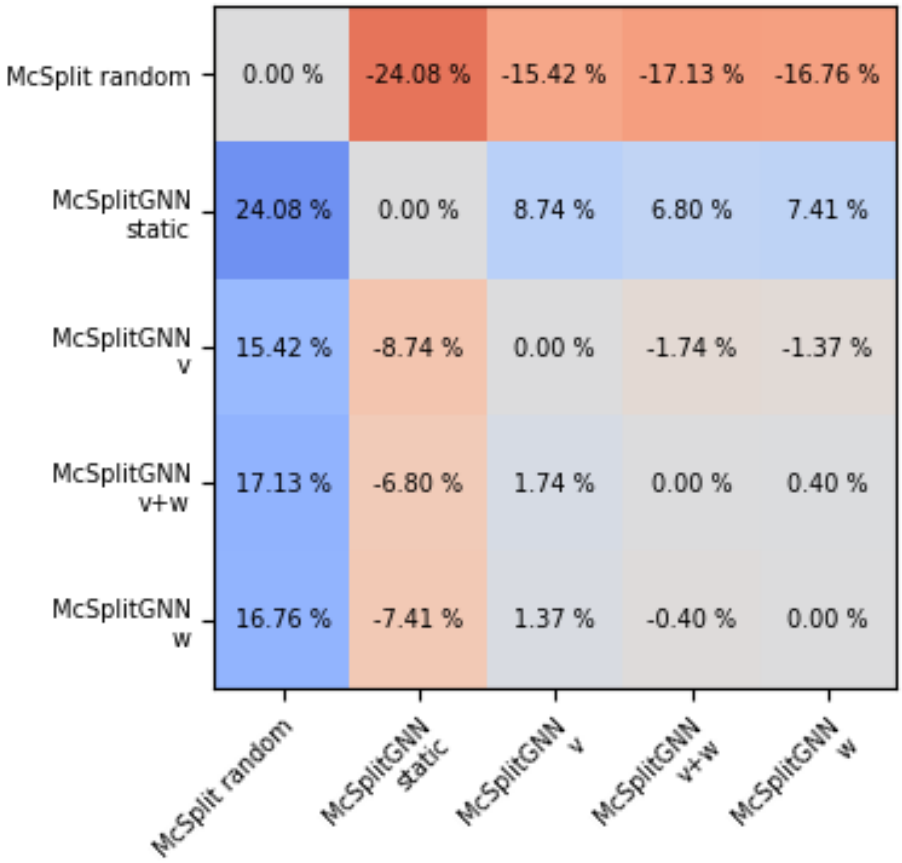

Figure 5.4: Average performance gain heatmap of McSplitGNN variants on *big finetuning* dataset

5.4.2 McSplitDiffGNN's performance

McSplitDiffGNN differs from McSplitGNN since it possesses only one GNN to choose the vertices from H, while the first part of the selection is left to our McSplitDAL+PR introduced in Chapter 3. Therefore, also the training is different since we have to train one GNN instead of two, so we increased the number of epochs to 5.

We can see our results compared to the random baseline in Figure 5.5 and Figure 5.6. Even these methods perform better than a random choice heuristic, allowing us to say our models learned a valid relationship. Moreover, we can see that the model trained on real data performs similarly to the one trained on synthetic data.

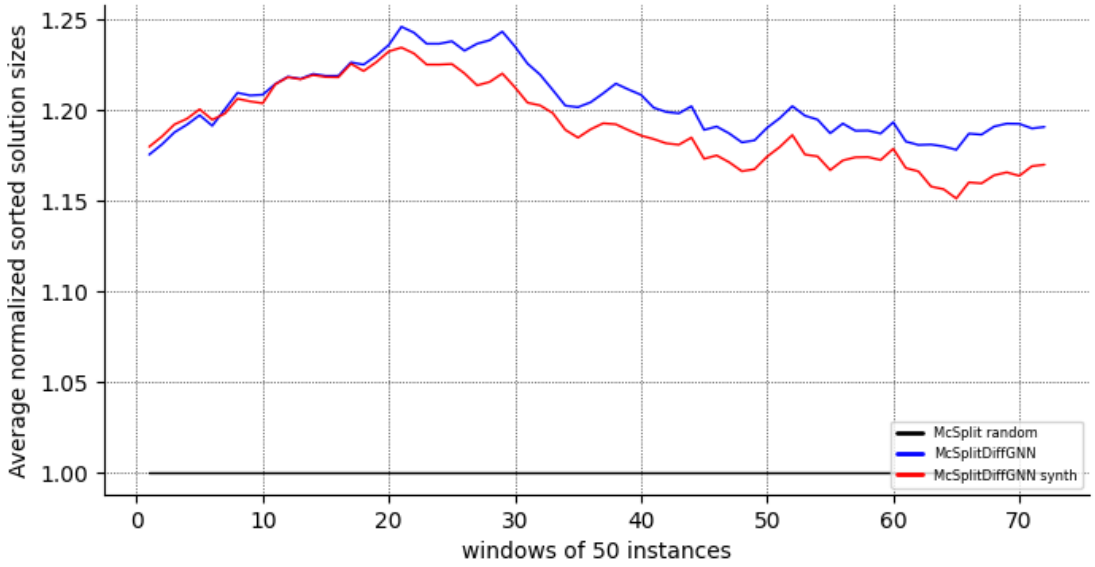

Figure 5.5: Rolling average comparison of McSplitDiffGNN variants on *big finetuning* dataset

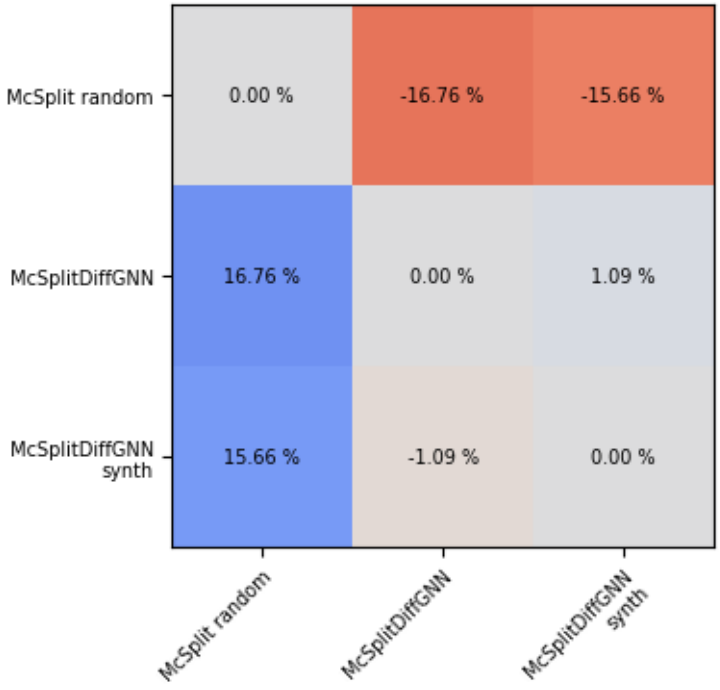

Figure 5.6: Average performance gain heatmap of McSplitDiffGNN variants on *big finetuning* dataset

5.4.3 Results

Our last comparison is between all of our techniques using Graph Neural Networks, using the random McSplit as baseline once again, but on the whole *big* dataset.

We show both the line plot with rolling average (Figure 5.7) and the heatmap of gains (Figure 5.8). We can see that all the presented techniques are better than the random one, as we already validated, and that the most performing one is the McSplitGNN static since it consistently stays above the other curves in the line plot, and the heatmap confirms it too.

Our McSplitGNN with both GNNs active is the second most performant technique, while in the third place, we have McSplitGNN using only the right GNN and McSplitDiffGNN synth. This draw and the fact that McSplitGNN using both GNNs is more performant than them allow us to make two remarks. Firstly, when we use only the right GNN, the performance seems to be mainly influenced by the left vertex of the pair, chosen by our McSplitDAL. It is evident since we get almost identical performance using two different architectures and two different training procedures. Secondly, a hybrid approach performs worse than one purely based on GNNs. We suppose that this result is due to an unbalanced mixture. The algorithm is unable to exploit both techniques at the same time and obtains smaller solutions.

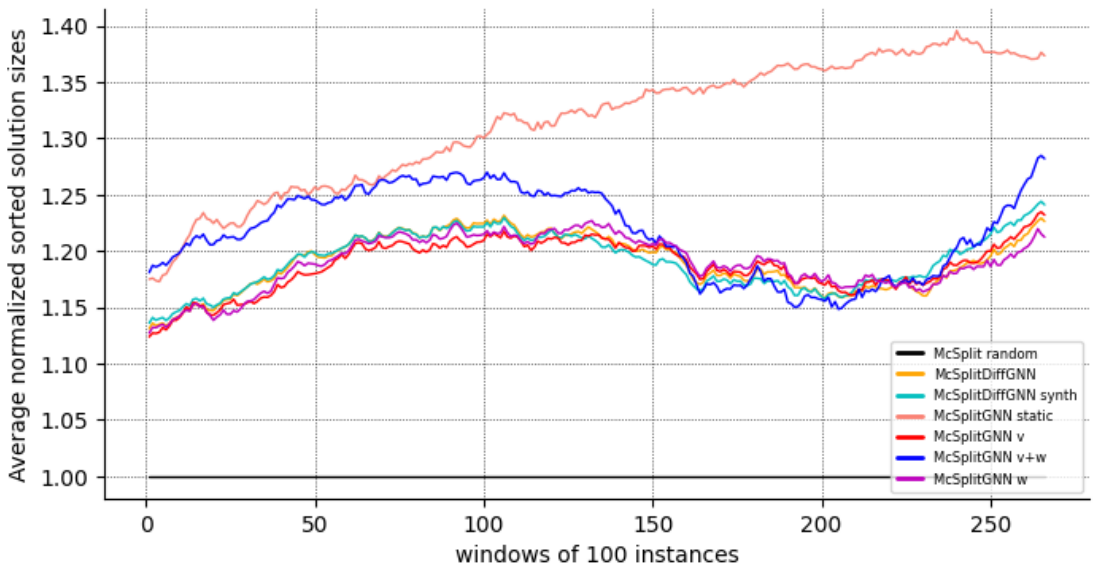

Figure 5.7: Rolling average comparison of GNN-based methods on *big* dataset

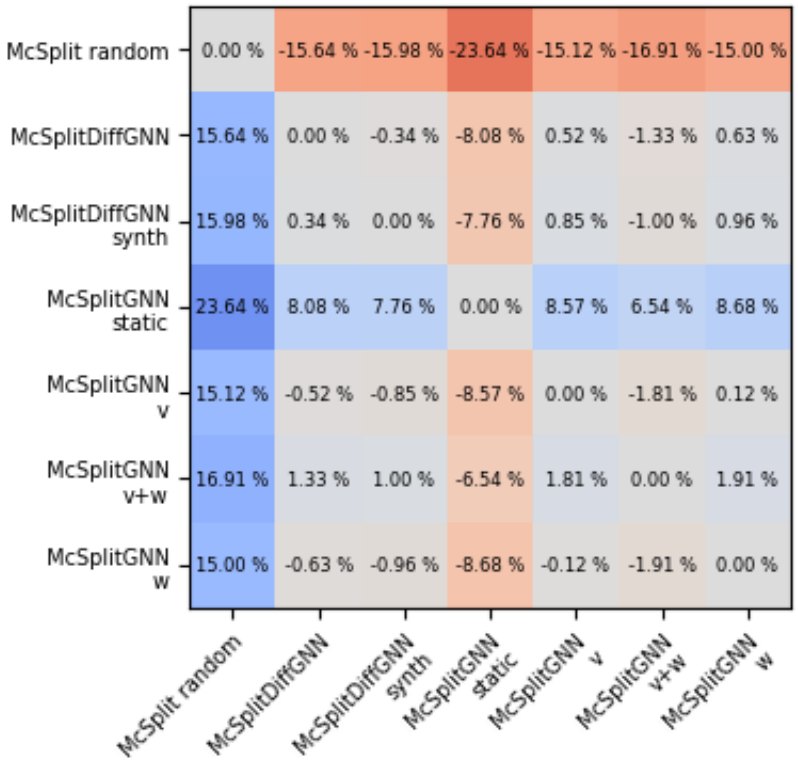

Figure 5.8: Average performance gain heatmap of GNN-based methods on *big* dataset

In the next and last chapter, we will draw our conclusions, comparing all the proposed methods of this thesis with McSplit's original algorithm.

Chapter 6

Conclusions

In the past chapters, we presented three classes of techniques: one focused on enhancing McSplit with new heuristics, one exploiting parallelism, and one harnessing the power of Graph Neural Networks. In this last chapter, we will analyze our datasets and draw some conclusions about our methods, comparing them against the standard baseline for MCIS: McSplit's original algorithm. Finally, we will propose some future works regarding our strategies.

6.1 Dataset analysis

In our work, we showed our results on the *small* dataset and on the *big* dataset.

The former contains 400 test instances made by pairs of graphs with no more than 100 nodes. Figure 6.1 shows some statistics about this dataset, plotting the average size of a pair and a heatmap showing the size of the coupled graphs. We can notice that almost all graphs contain 100 nodes, while a few are much smaller (between 10 and 50 nodes). As we can see from the second plot, all pairs include graphs of the same size. However, these test instances have different percentages of connectivity, guaranteeing a reasonable degree of variability.

The latter dataset, instead, represents Autonomous Systems constructed from the Border Gateway Protocol logs and is part of a collection of datasets ([17]). Differently from the *small* one, this dataset contains graphs of variable size, ranging from slightly less than 500 to more than 6000 nodes. We plotted its statistics too, as we can see from Figure 6.2, and we can observe that pairs are much more differentiated and the average pair size has a more even distribution.

To draw our conclusions about the methods we presented in this work, we employ both datasets since each of them offers variability, either in terms of connectivity or thanks to a wide range of different-sized pairs.

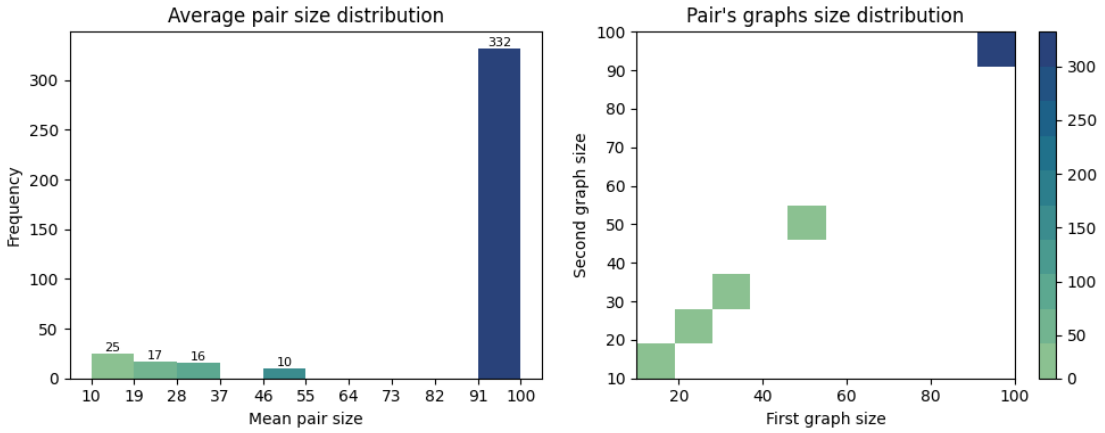

Figure 6.1: Statistics of *small* dataset

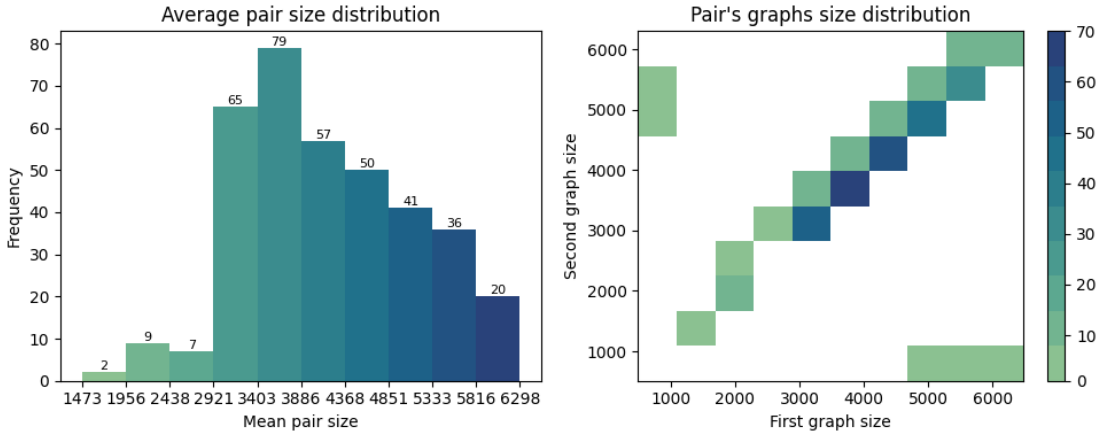

Figure 6.2: Statistics of *big* dataset

6.2 Results

We proposed five new methods to face the MCIS problem, each having a few variants by changing its parameters (like the heuristic, for example). Naturally, it would be unfeasible to compare all the possible combinations, so we treated them separately in their chapter.

Thanks to the comparisons we already performed, we can select the best variant for each proposed method and compare them, using as a baseline the original McSplit. Our best candidates are: McSplitMB 32T + KC*, McSplitBS 32T 32B + PR without DAL, McSplitDAL + PR, McSplitGNN static, and McSplitDiffGNN synth.

We run them on both *small* and *big* datasets, employing the original McSplit as a baseline.

We can see the results on *big* dataset in Figure 6.3 and Figure 6.4, respectively, using a line plot with rolling average and the heatmap of performance gains. All our methods provide an improvement over the base McSplit, except for the ones based on Graph Neural Networks.

We can find a very similar pattern on the *small* dataset (Figure 6.5 and Figure 6.6), hence strengthening our claims.

Specifically, our McSplitMB is the best method overall, followed by McSplitBS and our McSplitDAL. McSplitGNN used as a static heuristic seems to be beneficial on larger graph instances: we can see this pattern at the right of Figure 6.3 (we ordered results by ascending size of the solution).

This observation is encouraging since one of the main goals behind GNN-based methods was to overcome the limitations of standard McSplit when graphs become too big. Lastly, the McSplitDiffGNN synth is the least performant among the proposed techniques. Its bad performance is probably due to its hybrid nature since it combines McSplitDAL's RL framework and a GNN for choosing vertices from the second graph, being unable to exploit them at their fullest.

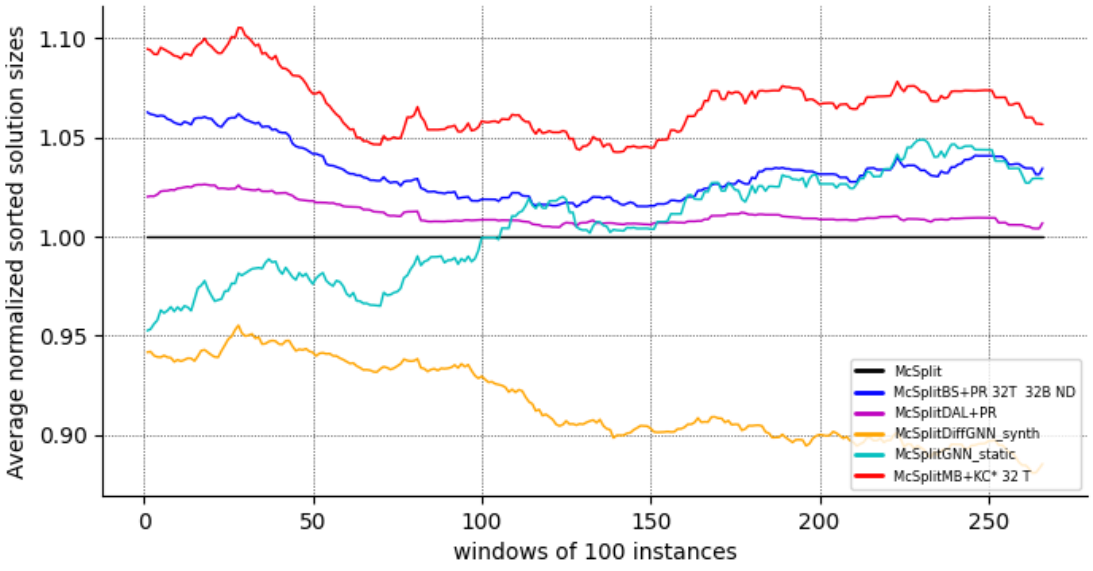

Figure 6.3: Rolling average comparison of the best methods on *big* dataset

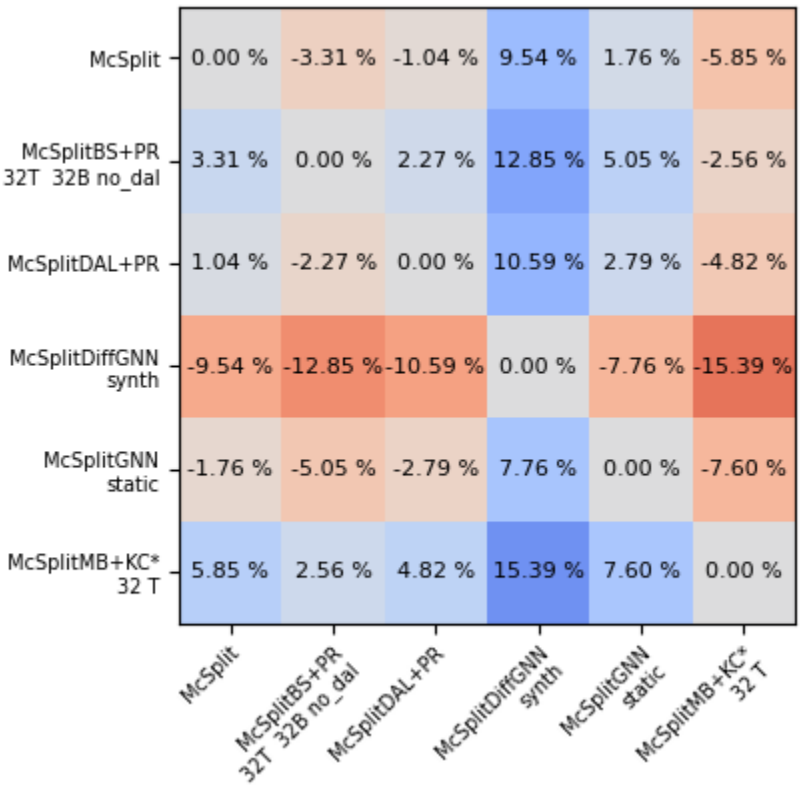

Figure 6.4: Average performance gain heatmap of the best methods on *big* dataset

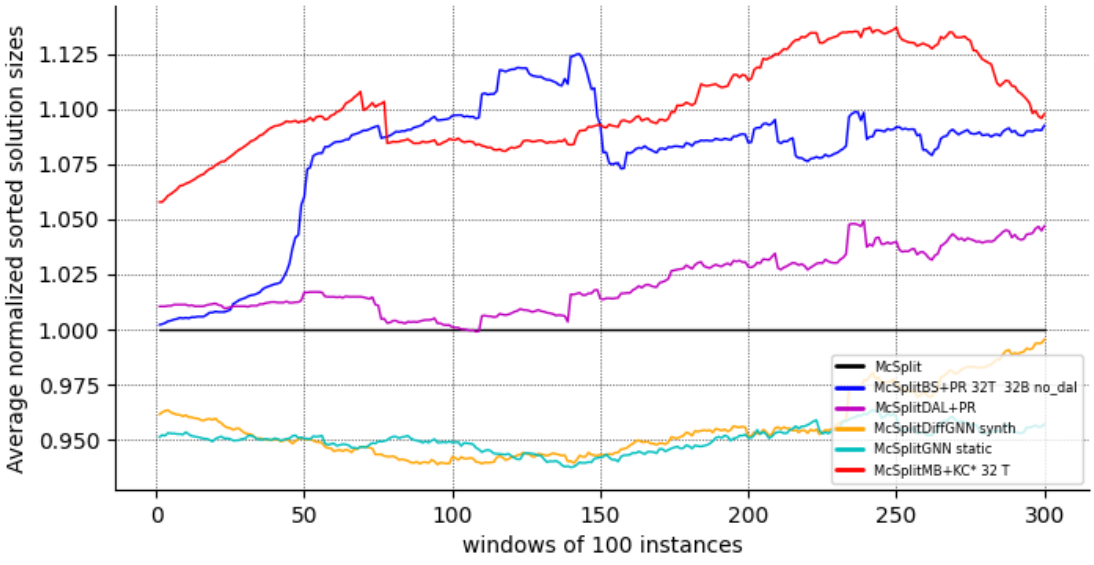

Figure 6.5: Rolling average comparison of the best methods on *small* dataset

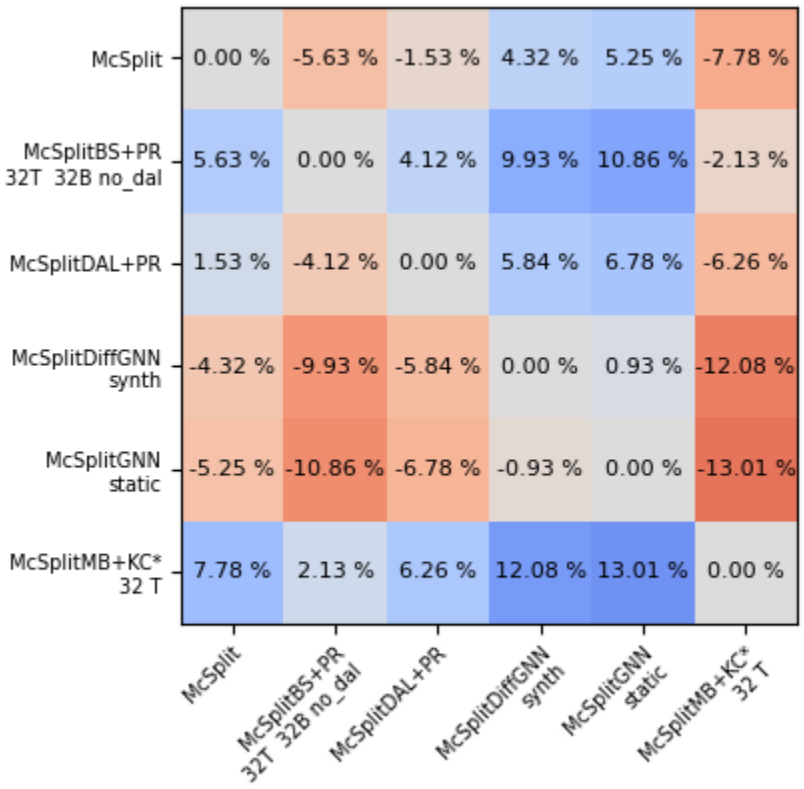

Figure 6.6: Average performance gain heatmap of the best methods on *small* dataset

Table 6.1 summarizes the results, showing the gain of each proposed method over the McSplit original algorithm. Notice that these improvements are relative to the baseline and do not provide insights on other methods' relative performance, which are present in the previous heatmaps.

Method	Gain on *small*	Gain on *big*
McSplitBS 32T 32B + PR without DAL	5.63%	3.31%
McSplitDAL + PR	1.53%	1.04%
McSplitDiffGNN synth	-4.32%	-9.54%
McSplitGNN static	-5.25%	-1.76%
McSplitMB 32T + KC*	**7.78%**	**5.85%**

Table 6.1: Average performance gain of the best methods over original McSplit

6.3 Future works

We analyzed all the presented methods from a performance point of view, but there are many possible improvements worth trying, which we will leave as potential future works.

Our McSplitDAL led to publishing a paper at ICSOFT 2023 (18th International Conference on Software Technologies), and it is also our most perfected method.

Concerning our parallel implementations, McSplitMB could improve by including techniques like LUM, McSplitSwap, and Reinforcement Learning. On the other side, our McSplitBS can be enhanced both in terms of execution speed and also in the management of the shared global queue.

Finally, the methods based on Graph Neural Networks have the most room for improvement. Right now, these methods are the worst in terms of performance, but it is necessary to remind they were trained for a reduced timeframe and on a limited amount of data. Moreover, it is possible to refine the learning procedure and the architecture to model more complex relationships.

Contributions

This work has been carried out jointly with another student, Marco Porro. We both contributed to the methods reported in this thesis, by sharing the workload. More specifically, my main contributions were the following:

- Chapter 3: McSplitDAL implementation, management of isolated rewards, initialization and heuristics
- Chapter 4: McSplit Multi-branch version with adjacency lists, McSplitBS optimization, no dal variant, and concurrency management
- Chapter 5: Reformatting of GLSearch and model analysis, implementation of McSplitDiffGNN testing procedure, training on the synthetic dataset

The remaining part of this work was carried out independently.

Acronyms

BC Betweenness Centrality

BS Branch-Sharing

CC Closeness Centrality

DAL Domain Action Learning

DQN Deep Q Networks

GNN Graph Neural Networks

KC Katz Centrality

LCC Local Clustering Coefficient

MB Multi-Branch

MCIS Maximum Common Induced Subgraph

MCS Maximum Common Subgraph

PR PageRank

RL Reinforcement Learning

Bibliography

[1] Ciaran McCreesh, Patrick Prosser, and James Trimble. «A Partitioning Algorithm for Maximum Common Subgraph Problems». In: (2017), pp. 712–719. DOI: 10.24963/ijcai.2017/99. URL: https://doi.org/10.24963/ijcai.2017/99 (cit. on pp. 1, 3, 6).

[2] Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. «GLSearch: Maximum Common Subgraph Detection via Learning to Search». In: *Proc. of Thirty-eighth International Conference on Machine Learning (ICML'21)*. 2021 (cit. on pp. 2, 3, 47).

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. «Playing atari with deep reinforcement learning». In: *arXiv preprint arXiv:1312.5602* (2013) (cit. on p. 11).

[4] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. «A Gentle Introduction to Graph Neural Networks». In: *Distill* (2021). DOI: 10.23915/distill.00033 (cit. on p. 11).

[5] Ameya Daigavane, Balaraman Ravindran, and Gaurav Aggarwal. «Understanding Convolutions on Graphs». In: *Distill* (2021). DOI: 10.23915/distill.00032 (cit. on p. 12).

[6] Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. «A Learning Based Branch and Bound for Maximum Common Subgraph Related Problems». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.03 (2020), pp. 2392–2399. DOI: 10.1609/aaai.v34i03.5619. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5619 (cit. on p. 13).

[7] Jianrong Zhou, Kun He, Jiongzhi Zheng, Chu-Min Li, and Yanli Liu. *A Strengthened Branch and Bound Algorithm for the Maximum Common (Connected) Subgraph Problem*. 2022. arXiv: 2201.06252 [cs.DS] (cit. on p. 14).

[8] Yanli Liu, Jiming Zhao, Chu-Min Li, Hua Jiang, and Kun He. *Hybrid Learning with New Value Function for the Maximum Common Subgraph Problem*. 2022. arXiv: 2208.08620 [cs.AI] (cit. on pp. 14, 21).

[9] James Trimble. «Partitioning algorithms for induced subgraph problems». PhD thesis. University of Glasgow, 2023 (cit. on pp. 16, 21).

[10] Sergey Brin and Lawrence Page. «The anatomy of a large-scale hypertextual Web search engine». In: *Computer Networks and ISDN Systems* 30.1 (1998). Proceedings of the Seventh International World Wide Web Conference, pp. 107–117. ISSN: 0169-7552. DOI: https://doi.org/10.1016/S0169-7552(98)00110-X. URL: https://www.sciencedirect.com/science/article/pii/S016975529800110X (cit. on p. 17).

[11] Alex Bavelas. «Communication patterns in task-oriented groups». In: *The journal of the acoustical society of America* 22.6 (1950), pp. 725–730 (cit. on p. 18).

[12] Duncan J Watts and Steven H Strogatz. «Collective dynamics of 'small-world' networks». In: *nature* 393.6684 (1998), pp. 440–442 (cit. on p. 18).

[13] Linton C Freeman. «A set of measures of centrality based on betweenness». In: *Sociometry* (1977), pp. 35–41 (cit. on p. 18).

[14] Leo Katz. «A new status index derived from sociometric analysis». In: *Psychometrika* 18.1 (1953), pp. 39–43 (cit. on p. 20).

[15] P. Foggia, C. Sansone, and M. Vento. «A Database of Graphs for Isomorphism and Sub-Graph Isomorphism Benchmarking». In: -. Jan. 1, 2001, pp. 176–187 (cit. on p. 21).

[16] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. «Graphs over time: densification laws, shrinking diameters and possible explanations». In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 2005, pp. 177–187 (cit. on p. 21).

[17] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014 (cit. on p. 59).