POLITECNICO DI TORINO

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

Master's Degree Thesis

Designing new Maximum Common Subgraph solvers: Heuristics, Multi-Threading and Graph Neural Networks

Supervisor

Prof. Stefano QUER

Candidate

Marco PORRO

July 2023

Abstract

The Maximum Common Subgraph (MCS) is a complex theoretical computer science problem, generalization of the Subgraph Isomorphism problem, finding applications in diverse domains such as chemistry, biology, medicine, and network management. This research focuses on improving the performance of the McSplit algorithm, a popular recursive Branch-and-Bound MCS solver. The objective of this project is to develop a tool that efficiently identifies the largest subgraph within a given time limit, addressing real-world scenarios where finding the optimal solution is not always the primary goal.

To achieve this, a diverse range of strategies have been explored to identify the most effective approaches for future advancements. Three distinct implementations have been developed as part of this effort. Firstly, novel sorting heuristics have been devised for the McSplit algorithm, aiming to establish a best-first vertex selection policy during the tree search. These heuristics have been combined with a newly designed multi-thread parallel architecture, optimizing the allocation of processor time to promising branches of the recursive algorithm. Additionally, Graph Neural Networks (GNNs) have been employed to identify and prioritize the most favorable branches at each intersection of the tree search.

To assess the performance of these tools, extensive testing has been conducted using open-source datasets pertaining to Internet infrastructure networks and other relevant real-world applications. The evaluation of performance metrics demonstrates significant improvements over existing state-of-the-art MCS solvers in the targeted use cases. In conclusion, this research project has successfully enhanced the capabilities of the McSplit algorithm, providing notable advancements in solving the Maximum Common Subgraph problem for practical applications. These achievements have been recognized through the acceptance of our paper titled *A web scraping algorithm to enhance maximum common subgraph computation* at ICSOFT 2023. Furthermore, the analysis of the proposed methodologies, including sorting heuristics, parallel architecture, and Graph Neural Networks, offer valuable insights for future research to determine the most promising strategies among the explored approaches.

Acknowledgements

This milestone marks the end of an incredibly rewarding double-degree program between Politecnico di Torino and the University of Illinois at Chicago (UIC).

I am deeply thankful to my advisor, Prof. Stefano Quer, and the doctoral candidates, Andrea Calabrese and Lorenzo Cardone, for their invaluable assistance and availability. A heartfelt appreciation goes to my UIC advisor, Prof. Abolfazl Asudeh, for his guidance and support.

I am indebted to my fellow student colleagues, starting with my thesis partner, Salvatore Licata, for their collaboration and support. I am also grateful to the Chicago group and to my long-standing university friends for their affection and companionship throughout this academic adventure.

Above all, I want to extend my heartfelt thanks to my family, who have always been there for me and supported me in countless ways. Their unwavering belief in me has made this incredible journey possible, and who knows how many more to come.

MP

Table of Contents

List of Tables viii

List of Figures ix

Acronyms xiii

1 Introduction 1

1.1 Objectives . . . 1
1.2 Thesis Structure . . . 2

2 Background 4

2.1 Graphs . . . 4
2.1.1 Definitions . . . 4
2.1.2 Graph isomorphism . . . 6
2.1.3 Subgraph isomorphism . . . 7
2.2 Maximum Common Subgraph (MCS) problem . . . 7
2.3 McSplit algorithm . . . 9
2.3.1 Existing MCS solvers . . . 9
2.3.2 McSplit . . . 9
2.3.3 McSplitSO and McSplitSD . . . 13
2.4 Reinforcement Learning (RL) . . . 14
2.4.1 Theoretical overview . . . 14
2.4.2 An example of Q-learning . . . 15
2.4.3 McSplitRL . . . 16
2.4.4 McSplitLL . . . 17
2.5 Machine Learning meets graphs: GNNs . . . 19
2.5.1 Multi-Layer Perceptrons (MLPs) . . . 19
2.5.2 Graph Neural Networks (GNNs) . . . 22
Graph Convolutional Networks (GCNs) . . . 22
Graph Attention Networks (GATs) . . . 23

3 Algorithmic Optimizations and Heuristics 24
3.1 McSplitDAL . . . 25
3.1.1 Overview . . . 25
Domain Action Learning . . . 25
Hybrid learning policy . . . 25
3.1.2 Our implementations . . . 27
Joint vs Isolated Q-tables . . . 27
Initialization of Q-tables . . . 27
McSplitRL+DAL vs McSplitLL+DAL . . . 28
3.2 Static Heuristics . . . 29
3.2.1 PageRank (PR) . . . 31
3.2.2 Betweenness Centrality (BC) . . . 33
3.2.3 Closeness Centrality (CC) . . . 35
3.2.4 Katz Centrality* (KC*) . . . 35
3.2.5 Local Clustering Coefficient (LCC) . . . 38
3.2.6 Summary . . . 40

4 Parallel Architectures and Multi-Threading 41
4.1 McSplit Multi Branch (McSplitMB) . . . 42
4.2 McSplit Branch Sharing (McSplitBS) . . . 43
4.2.1 Building an iterative version of McSplit . . . 44
4.2.2 Branch Sharing . . . 45
Block size . . . 48
Delayed Sharing . . . 50
4.3 Conclusion . . . 51

5 Graph Neural Networks (GNN) 53
5.1 GLSearch . . . 54
5.1.1 The architecture . . . 54
5.1.2 Training . . . 55
5.1.3 Our experience . . . 57
5.2 McSplitGNN . . . 57
5.2.1 The architecture . . . 58
5.2.2 Training . . . 59
5.3 McSplit DiffGNN . . . 60
5.3.1 Model Architecture . . . 60
5.3.2 Training . . . 61
5.3.3 Training on synthetic data . . . 62

6 Experimental Analysis 64
6.1 The experimental setup . . . 64
6.1.1 Testing methodology . . . 64
6.1.2 Datasets . . . 66
6.1.3 Result post-processing . . . 68
Gain plots . . . 69
Mean Normalized Difference (MND) . . . 69
6.2 Experimental Analysis of the Static Heuristics . . . 71
6.2.1 McSplitDAL implementations . . . 71
McSplitSD . . . 71
Joint vs Isolated Q-tables . . . 72
Initialization of Q-tables . . . 73
McSplitRL+DAL vs McSplitLL+DAL . . . 74
Comparison of all McSplitX variants . . . 74
6.2.2 A first toe in the water with PageRank . . . 75
6.2.3 Comparison of the static heuristics . . . 77
6.3 Multi-Threading architectures . . . 81
6.3.1 McSplit MultiBranch (MB) . . . 81
Thread count . . . 81
Static Heuristics . . . 83
6.3.2 McSplit Branch Sharing (BS) . . . 84
Block Size . . . 84
Thread Count . . . 86
Static Heuristics . . . 87
Delayed Sharing . . . 88
Is Reinforcement Learning effective in McSplitBS? . . . 89
6.3.3 Conclusions on Multi-Threaded McSplit . . . 90
6.4 GNN models . . . 93
6.4.1 McSplitGNN . . . 93
6.4.2 McSplit DiffGNN . . . 95
6.4.3 Conclusions on the GNN-based models . . . 97
6.5 Summary of Results . . . 98
6.5.1 Qualitative comments of the results . . . 98
6.5.2 Quantitative comparison of the best algorithms . . . 99

7 Conclusion 103

Contributions 105

A Additional Charts 107

Bibliography 115

List of Tables

2.1 Assignment of bidomain labels in McSplit 12

3.1 Static heuristics for McSplit 40

List of Figures

2.1 Basic graph types 5
2.2 Example of the MCS problem 8
2.3 Example of the McSplit algorithm during the recursive search . . . 11
2.4 Example of Q-learning 15
2.5 Leaf Vertex Union Match 18
2.6 A perceptron with n inputs 20
2.7 Multi-Layer Perceptron (MLP) 20
2.8 Popular activation functions 21
2.9 Graph Convolutional Network (GCN) 22

3.1 Example of Domain Action Learning 26
3.2 Example of Local Clustering Coefficient (LCC) 38

4.1 Branching structure of McSplit 42
4.2 McSplitMB 43
4.3 McSplitBS: iterative implementation of the recursive search 45
4.4 McSplitBS: global stack after T_0 reaches the termination condition . 47
4.5 McSplitBS: subdivision in threads 48
4.6 Comparison of broad vs focused exploration patterns 51

5.1 GLSearch architecture 55
5.2 McSplitGNN architecture 58
5.3 DiffGNN architecture 61

6.1 Progression of the best solution size ($|S|$) in McSplit 65
6.2 Statistics of the Large dataset 67
6.3 McSplit performance on Small dataset 68
6.4 Comparison of McSplitDAL and McSplitDAL-SD on Small 71
6.5 Comparison of McSplitDAL Joint and McSplitDAL Isolated on Small 72
6.6 Comparison of McSplitDAL and McSplitDAL (init) on Small . . . 73
6.7 Comparison of McSplitLL+DAL and McSplitRL+DAL on Small . 74
6.8 Comparison of all McSplitX variants on Small 75

6.9 Comparison of all McSplitX+PR variants on Small 76
6.10 Analysis of outliers in McSplitX+PR variants on Small 77
6.11 Comparison of all McSplitDAL+ variants on Small 78
6.12 Comparison of all McSplitDAL+ variants on Large 79
6.13 Comparison of McSplitMB+PR using a different number of threads on Large-Finetuning 82
6.14 Comparison of McSplitMB versions using 32 threads and different static heuristics on Large-Finetuning 84
6.15 Comparison of McSplitBS versions using PageRank, 32 threads, and different block sizes on Large-Finetuning 85
6.16 Comparison of McSplitBS versions using PageRank, 32 block size, and different thread counts on Large-Finetuning 86
6.17 Comparison of McSplitBS versions using 32 threads, a block size of 32, and different static heuristics on Large-Finetuning 88
6.18 Comparison of McSplitBS versions with or without delayed sharing using 32 threads, a block size of 32, and different heuristics on Large-Finetuning 89
6.19 Comparison of McSplitBS versions with and without DAL policy, using 32 threads, a block size of 32, and different heuristics on Large-Finetuning 90
6.20 Comparison of the best performing versions of McSplitMB and McSplitBS using 32 threads, a block size of 32, and different heuristics on Large 91
6.21 Comparison of all McSplitGNN and McSplitGNN static on Large-Finetuning 94
6.22 Comparison of McSplit DiffGNN and DiffGNN synthetic on Large-Finetuning 96
6.23 Comparison of the best-performing algorithms for each family on Large 100
6.24 Comparison of the best-performing algorithms for each family on Small 102

A.1 Full difference heatmap of McSplitBS versions using PageRank, 32 threads, and different block sizes on Large-Finetuning 108
A.2 Full comparison of McSplitBS versions using delayed sharing or not, with 32 threads, block size of 32, and different heuristics on Large-Finetuning 109
A.3 Full difference heatmap of McSplitBS versions using delayed sharing or not, with 32 threads, block size of 32, and different heuristics on Large-Finetuning 110

A.4 Full comparison of McSplitBS versions using the DAL policy or not, with 32 threads, block size of 32, and different heuristics on Large-Finetuning . . . 111
A.5 Full difference heatmap of McSplitBS versions using the DAL policy or not, with 32 threads, block size of 32, and different heuristics on Large-Finetuning . . . 112
A.6 Full comparison of McsplitMB and McSplitBS, with 32 threads, block size of 32, and different heuristics on Large . . . 113
A.7 Full difference heatmap of McsplitMB and McSplitBS, with 32 threads, block size of 32, and different heuristics on Large . . . 114

Acronyms

BC
: Betweenness Centrality

BFS
: Breadth First Search

BS
: Branch Sharing

CC
: Closeness Centrality

DAL
: Domain Action Learning

DFS
: Depth First Search

DQN
: Deep Q-Network

FSM
: Finite State Machine

GAT
: Graph Attention Network

GCN
: Graph Convolutional Network

GNN

Graph Neural Network

KC

Katz Centrality

LCC

Local Clustering Coefficient

LSM

Long-Short term Memory

LUM

Leaf vertex Union Match

MB

Multi Branch

MCCS

Maximum Common Connected Subgraph

MCES

Maximum Common Edge Subgraph

MCIS

Maximum Common Induced Subgraph

MCS

Maximum Common Subgraph

MDP

Markov Decision Process

ML

Machine Learning

MLP

Multi-Layer Perceptron

MND

Mean Normalized Difference

MSE

Mean Squared Error

NN

Neural Network

NP

Non-deterministic Polynomial-time

PDF

Probability Density Function

PR

PageRank

PTAS

Polynomial-time Approximation Scheme

RL

Reinforcement Learning

SD

Swapping of Density

Chapter 1

Introduction

The field of theoretical computer science plays a fundamental role in advancing our understanding of computational problems and their complexity. Within this domain, the MAXIMUM COMMON SUBGRAPH (MCS) problem stands as a classic and challenging conundrum with numerous real-world applications, stemming from software engineering to structural biology and computational chemistry. The MCS problem seeks to identify the largest possible subgraph shared between two given graphs, offering insights into graph similarity and their structural relationships.

Given the elevated computational complexity of the MCS problem, a significant body of research has been focused on developing more efficient algorithms that can solve it in a faster and more scalable manner. The inherent difficulty of the problem stems from its NP-hard nature, which implies that finding an optimal solution within a reasonable amount of time becomes increasingly challenging as the input graphs grow in size and complexity. As a result, researchers are actively exploring algorithmic advancements, optimization techniques, and parallel computing approaches to improve the efficiency and scalability of MCS algorithms. These efforts aim to reduce the computational burden associated with the MCS problem, improving its effectiveness in larger-scale graph analysis tasks and facilitating its integration into real-world applications.

1.1 Objectives

The primary objective of this thesis is to enhance the performance of one of the currently most popular MCS ground-truth solvers, McSplit. Due to the broad and comprehensive nature of our goals, we opted for a breadth-first search approach for our work, exploring and implementing a wide range of improvement techniques, analyzing their effectiveness and identifying the strategies that yield promising results. This methodology allows us to explore different avenues for enhancing

McSplit, without getting too deeply entrenched in any single strategy. It enables us to survey a broader landscape of potential improvements and compare the performance of various approaches across different scenarios.

Throughout our exploration, we will evaluate different algorithmic enhancements, including heuristic node priority, parallel multithreaded architectures and machine learning, aiming to improve the quality of the produced solutions within a limited time frame.

Our goal is to identify strategies that can potentially improve upon the existing state-of-the-art solutions. If a particular strategy demonstrates a significant increase in runtime efficiency, it would represent a valuable advancement in the field. On the other hand, if a strategy fails to yield substantial improvements, we will critically analyze and explain the reasons behind its limitations, providing valuable insights to the scientific community.

Through rigorous experimentation and analysis, we will compare the performance of our modified versions of McSplit against the original algorithm and other state-of-the-art MCS solvers. By benchmarking the different implementations on a large set of test instances, including real-world datasets, we can draw meaningful conclusions about the effectiveness of each strategy and provide empirical evidence to support our findings.

1.2 Thesis Structure

The thesis is structured into several chapters that systematically explore different aspects of the MCS problem and its potential enhancements. Each chapter focuses on a specific area of investigation, contributing to the overall understanding and improvement of MCS solvers.

Chapter 2, "Background", delves into the fundamental concepts related to the MCS problem. It provides a comprehensive background on graphs, including definitions, properties, and common terminology. The chapter then dives into the theoretical foundations of the MCS problem, exploring concepts such as graph isomorphism and subgraph isomorphism. Furthermore, it introduces McSplit and discusses its variants that have been proposed over the years. To explore some of these variants, the chapter also covers Reinforcement Learning, related to the McSplitRL and McSplitLL algorithms, and a brief overview of Machine Learning and Graph Neural Networks (GNNs).

In Chapter 3, "Algorithmic Optimizations and Heuristics", the focus shifts to investigating different heuristics for improving MCS by prioritizing the best branches of the search space. The strengths, limitations, and potential trade-offs of each heuristic are thoroughly discussed, providing insights into their applicability and

impact on MCS problem-solving. Furthermore, the chapter explores our implementation of the current state-of-the-art MCS solver, McSplitDAL, including several potential optimizations that were introduced and tested during the development process.

Chapter 4, "Parallel Architectures and Multi-Threading", explores the potential benefits of parallelization in the context of MCS problem-solving. The chapter discusses the design and implementation of two multithreaded frameworks for McSplit, aiming to enhance its performance and scalability on multicore architectures. Moreover, the exploration of concurrent computing techniques provides a valuable platform to evaluate whether a deeply focused search approach can yield better results than a breadth-first strategy.

In Chapter 5, "Graph Neural Networks (GNN)", the focus turns to the application of GNNs as part of Machine Learning MCS solvers. The chapter proposes multiple GNN-powered models to guide the search process of McSplit, with the goal of improving its efficiency as a dynamic node priority heuristic. Special regard will be given to discussing the potential challenges and limitations of the proposed designs, as well as the applicability of GNNs to the problem at hand.

Chapter 6, "Experimental Analysis", presents a comprehensive analysis of the findings from the previous chapters. It summarizes the performance, strengths, and limitations of the explored strategies and techniques. Through comparative testing, the chapter identifies the most effective among the proposed MCS solvers. Furthermore, it discusses the broader implications of the research and potential future directions for enhancing MCS problem-solving techniques.

The thesis includes an appendix section that reports additional data and charts that were not included in the main body of the thesis.

Chapter 2

Background

In this section, we delve into key theoretical concepts relevant to the research work, and we define important terminology that will be used throughout the rest of the thesis. Additional concepts will be introduced in subsequent chapters as they become relevant.

2.1 Graphs

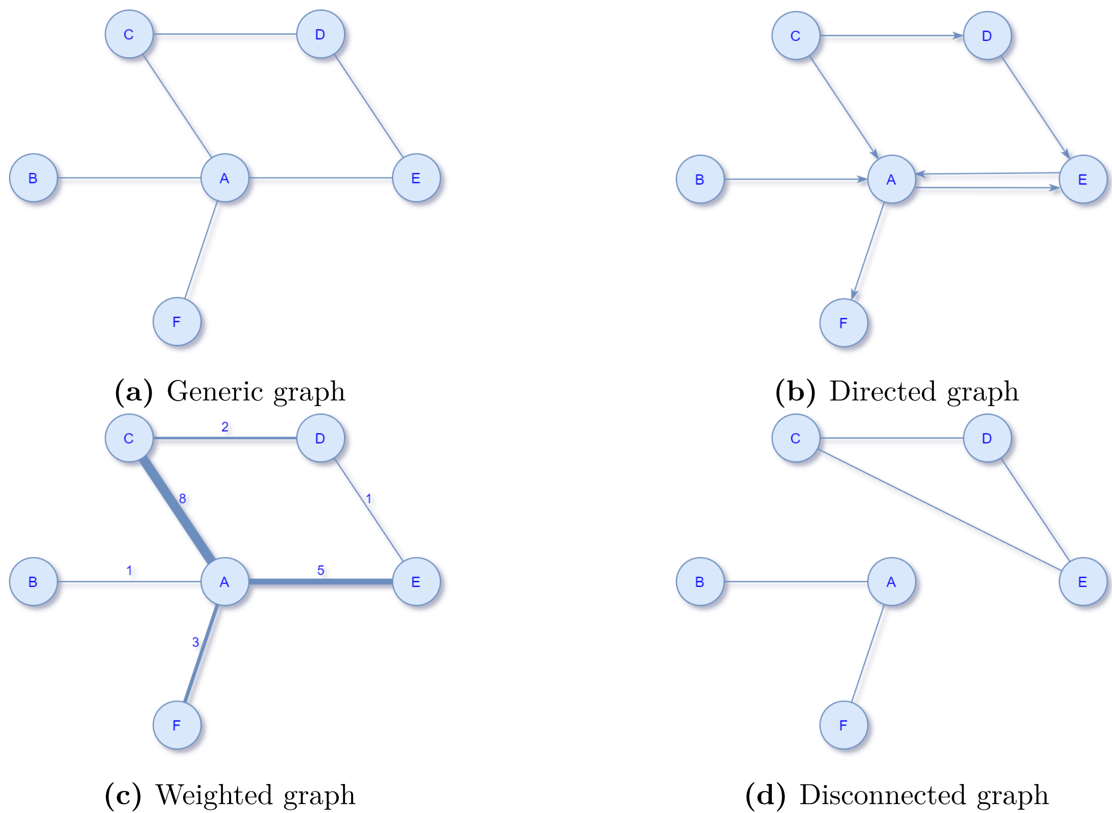
Graphs are mathematical structures that model the relationships between objects. Graph theory is the branch of mathematics that studies the properties and applications of graphs, which can be used to represent many real-world phenomena, such as social interactions, game logic, language systems, molecular structures, or infrastructure networks. Specifically, the field of graph theory is often considered to have been born from Leonhard Euler's Königsberg bridge problem in 1741 [1], which first used topological abstraction to decipher a real-world conundrum.

2.1.1 Definitions

We will now define the basic terminology used in graph theory:

Definition 1. A *Graph* is a mathematical object represented by $G(V, E)$, where V is a collection of entities, called vertices or nodes, and E is a set of relationships between pairs of entities, called edges.

Let $|V|$ represent the number of vertices in V, and $|E|$ denote the number of edges in E. For brevity, in this thesis we will occasionally use an abuse of notation to denote the number of vertices of a graph G as $|G|$. For two vertices u and v, if there exists an edge e connecting them, we can denote this edge as $e = (u, v)$. The vertices u and v are considered adjacent or neighbors, and e is incident to both



(a) Generic graph

(b) Directed graph

(c) Weighted graph

(d) Disconnected graph

Figure 2.1: Basic graph types

u and v. We call *neighborhood*(v) the set of nodes u such that $(u, v) \in E$. The degree $deg(v)$ of a vertex v corresponds to the number of edges incident to it.

An edge is classified as undirected if it models a bidirectional relationship between two vertices. Conversely, an edge is considered directed if it represents a relationship that is valid only in one direction. If any edge within a graph is directed, the graph as a whole is categorized as directed.

If in a graph G the edges can model relationships of different strengths, then G is said to be weighted, and each edge e is assigned a weight $w(e)$. Otherwise, G is said to be unweighted.

Two edges e_1 and e_2 are adjacent if they share a vertex u. A path is a continuous collection of edges, adjacent pair by pair, that connect two vertices u and v. The length of a path is the number of edges it contains. If in a graph G there exists a path between any two vertices, then G is said to be connected. Otherwise, G is said to be disconnected.

A graph G is said to be complete if every pair of vertices is connected by an

edge:

$$\forall u, v \in V, \qquad (u, v) \in E \tag{2.1}$$

If we consider a subset of vertices $\widetilde{V} \subseteq V$ that satisfies 2.1, then we can say that $\widetilde{V}$ forms a clique.

2.1.2 Graph isomorphism

We define the notion of graph isomorphism. The following definition considers only undirected, unweighted graphs, but the concept can be extended to other types of graphs as well.

Definition 2. Two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are said to be *isomorphic* if there exists a bijection $f : V_1 \to V_2$ such that for any two vertices $v_i, v_j \in V_1$, v_i and v_j are adjacent in G_1 if and only if $f(v_i)$ and $f(v_j)$ are adjacent in G_2.

Graph isomorphism plays a crucial role in graph theory as it enables us to compare two graphs independently of their specific representations or labels. This property is essential, as graphs are routinely stored and managed using different data structures, such as adjacency lists or adjacency matrices, each offering unique trade-offs in terms of memory usage and computational efficiency. Furthermore, the labels used to identify the vertices and edges may differ across graph instances. By determining if two graphs are isomorphic, we can establish their structural equivalence, which allows for the identification of common patterns, structures, or properties among different graphs, facilitating meaningful comparisons and analysis across various domains. Furthermore, graph isomorphism forms the basis for numerous graph algorithms and computational techniques, making it a fundamental concept with significant implications for graph analysis and problem-solving.

However, determining whether two graphs are isomorphic is a computationally expensive task, which has been extensively studied in theoretical computer science under the name of GRAPH ISOMORPHISM. Classified as a decision problem, it is believed to belong to the class of NP (Non-deterministic Polynomial-time) problems. While it remains an open question whether GRAPH ISOMORPHISM can be solved in polynomial time for all instances, significant progress has been made for specific classes of graphs, such as trees and planar graphs, for which the problem can be solved polynomially [2][3][4]. Moreover, it has been proven that under reasonable assumptions, GRAPH ISOMORPHISM is not NP-complete [5]. Due to its many both theoretical and practical applications, GRAPH ISOMORPHISM remains an active area of research, with ongoing efforts to develop more efficient algorithms and explore its underlying properties.

2.1.3 Subgraph isomorphism

Let us define the notion of subgraph. The following definition considers only undirected, unweighted graphs.

Definition 3. A graph $G' = (V', E')$ is said to be a *subgraph* of a graph $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E$. If a subgraph G' is *induced*, any two vertices $v_i, v_j \in V' \subseteq V$ are adjacent in G' if and only if they are also adjacent in G.

This definition resembles Definition 2. Specifically, we can postulate the following corollary:

Corollary 3.1. Two graphs G_1 and G_2 are isomorphic if and only if G_1 is an induced subgraph of G_2 and G_2 is an induced subgraph of G_1.

Therefore, we can consider the problem of determining whether a graph G_1 is an induced subgraph of another graph G_2 as a generalization of GRAPH ISOMORPHISM. This decision problem is known as SUBGRAPH ISOMORPHISM, and it is one of the cornerstones of theoretical computer science, as it generalizes other well-known problems such us MAXIMUM CLIQUE, HAMILTONIAN CYCLE and INDEPENDENT SET. As such, SUBGRAPH ISOMORPHISM is also proven to be NP-complete [6].

2.2 Maximum Common Subgraph (MCS) problem

Building upon the concept of graph isomorphism seen in Section 2.1, we can now delve into the subject of this research project: the MAXIMUM COMMON SUBGRAPH (MCS) problem. The MCS problem is an optimization problem that aims to find the largest subgraph that is common to two given graphs, and its decision version is consequently a further generalization of SUBGRAPH ISOMORPHISM. The problem is formally defined as follows:

Definition 4. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the MAXIMUM COMMON SUBGRAPH problem consists in finding a graph $G = (V, E)$ such that G is isomorphic to a subgraph of G_1 and to a subgraph of G_2, and $|V|$ is maximum.

Definition 4 can be trivially amended to support the concept of the Maximum Common Induced Subgraph (MCIS) problem, which specifically requires the subgraph G to be induced and strives to maximize the number of vertices in G. This is different from the MAXIMUM COMMON EDGE SUBGRAPH (MCES) where the graphs are not required to be induced, and the objective is to maximize the number of edges. However, the MCIS has much more relevance in the literature, to the

point that the terms MCS and MCIS are often used interchangeably. In this work, we will follow this convention and use the term MCS to refer to the MCIS problem.

Another common variant is the MAXIMUM COMMON CONNECTED SUBGRAPH (MCCS) problem. As the name implies, this problem requires the subgraph G to be connected, and it is consequently a specialized case of the MCS problem. In this work, we will focus on the more general MCS problem, as it is more difficult and the algorithms we will see can be easily adapted to solve the MCCS problem as well. Additionally, while our exploration of the MCS problem primarily revolves around undirected and unweighted graphs, the methodologies presented can be readily extended to encompass directed and weighted graphs.

MCS belongs to the class of NP-Hard [7]. Furthermore, it has been shown that it cannot be easily approximated, because it is of class MAX SNP-Hard [8], consequently it is not possible to find a PTAS (Polynomial-Time Approximation Scheme) for it unless $P = NP$.

This problem finds many practical applications in fields like bioinformatics [9], chemistry [10][11], software analysis [12] and open information extraction [13], and it is therefore of great interest to the scientific community. For this reason, many algorithms have been proposed to solve it, and we will see some of the most relevant ones in the next section.

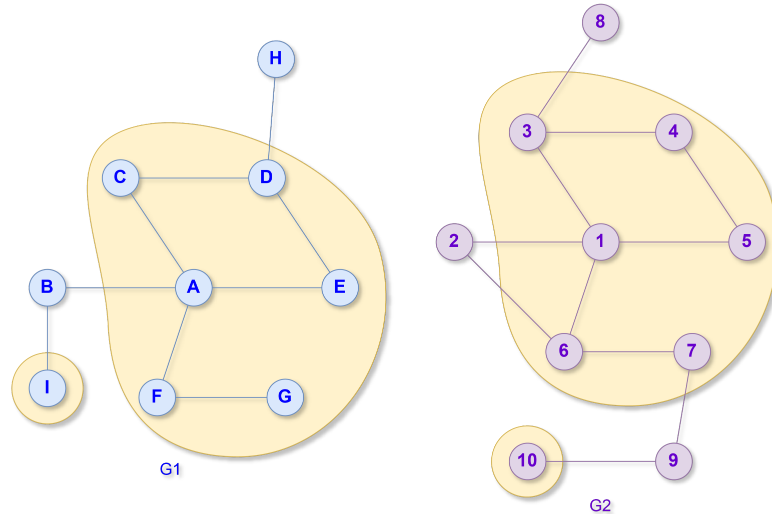


Figure 2.2: Example of the MCS problem

In 2.2 we can see an example of the MCS problem. The two graphs G_1 (blue) and G_2 (purple) have a common subgraph G (highlighted in yellow) of size 7. G can be expressed as a collection of matches of nodes of the two graphs, therefore we define the solution as a set S of matches $\langle u, v \rangle$ between the nodes $u \in V_1$ and

$v \in V_2$. In this case,

$$S = \{\langle A,1\rangle, \langle C,3\rangle, \langle D,4\rangle, \langle E,5\rangle, \langle F,6\rangle, \langle G,7\rangle, \langle I,10\rangle\}$$

If this solution S represents the largest common subgraph, then it is called the *optimal* solution of the MCS problem. Following this notation, any other induced subgraph of G_1 and G_2 with fewer nodes than S is still called a solution, but not an optimal one.

Note that G is not connected, and therefore it is not a solution to the MCCS problem. The optimal solution to the MCCS problem is the set $S' = S \setminus \{\langle I,10\rangle\}$

2.3 McSplit algorithm

2.3.1 Existing MCS solvers

The MCS problem can be approached using different types of algorithms. One category consists of ground-truth solvers, which aim to find the exact optimal solution. Prominent examples include McSplit [14] and $k\downarrow$ [15], or other methods that rely on clique-based approaches[16][17]. These solvers employ techniques such as recursive backtracking and constraint satisfaction to perform an exhaustive search of the solution space. As a result, they guarantee to find the optimal solution, albeit in a potentially lengthy but finite amount of time.

Another category of MCS algorithms comprises approximate graph-matching methods. These methods provide a faster alternative to exact solvers but may not necessarily yield an actual common induced subgraph. They can be further divided into supervised models, including I-PCA [18], GMN [19], and NeuralMCS [20], and unsupervised models like GW-QAP [21] and RLMCS [22]. Supervised models typically require labeled training data and employ machine learning techniques to approximate the solution. On the other hand, unsupervised models might use mixed approaches or leverage optimization algorithms to iteratively refine their approximation.

While approximate models offer computational efficiency, they often rely on exact solvers to generate labels for training and cannot guarantee the optimal solution. In this research project, our focus will be on the exact solver McSplit, which represents the current state-of-the-art MCS ground-truth solver. By investigating McSplit and exploring strategies to enhance its performance, we strive to directly or indirectly contribute to the advancement of all types of MCS-solving algorithms.

2.3.2 McSplit

McSplit is a backtracking algorithm that uses a branch-and-bound approach to solve the MCS problem. It was first introduced in 2017 by McCreesh et al. [14] and

has since been improved and optimized. The algorithm is based on the observation that the size of the largest common subgraph is bounded by the size of the smallest of the two given graphs. This observation is used to prune the search space and reduce the number of recursive calls, thus reducing the overall runtime, but still guaranteeing the optimal solution.

```
1  S ← {}
2
3  Function McSplit(G, H)
4  |   S = mcs(G, H, {})
5  |   return S
6
7  Function mcs(G, H, S_current)
8  |   if |S_current| > |S| then
9  |   |   S ← S_current
10 |   end
11 |   Bound ← ComputeBound(G, H, |S_current|)
12 |   if Bound < |S| then
13 |   |   return
14 |   end
15 |   β ← SelectBidomain(G, H)
16 |   v ← SelectVertexV(G, β)
17 |   G' ← G \ {v}
18 |   while w ∈ SelectVertexW(H, β) do
19 |   |   S'_current ← S_current ∪ {v : w}
20 |   |   H' ← H \ {w}
21 |   |   UpdateBidomains(G', H', v, w)
22 |   |   mcs(G', H', S'_current)
23 |   end
24 |   mcs(G', H, S'_current)
25 |   return
```

Algorithm 1: The McSplit algorithm.

In Algorithm 1 we show a simplified pseudocode of McSplit. The algorithm takes two graphs $G(V_G, E_G)$ and $H(V_H, E_H)$ as input and returns the MCS S. During the recursive search, S will hold the best solution found until that point, while $S_{current}$ will hold the induced subgraph identified by the current search branch. Both S and $S_{current}$ are saved as sets of node matches $\langle v, w \rangle$, but they logically represent a graph. In this algorithm, as well as the rest of this thesis, we will consider v a generic node of G and w a generic node of H. The algorithm starts by initializing S to the empty set (line 1), then it immediately calls the recursive

function *mcs* (line 4) passing as initial current solution another empty set.

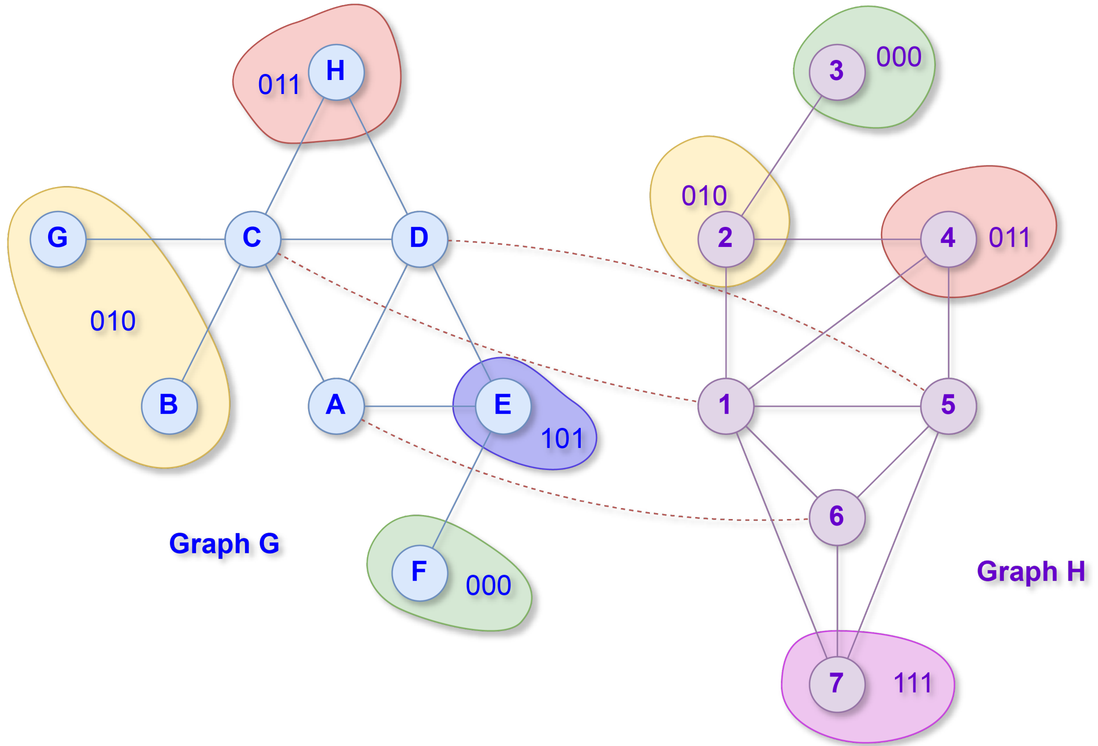


Figure 2.3: Example of the McSplit algorithm during the recursive search

The function *mcs* uses the concept of *Bidomain* to select the next pair of vertices to be added to $S_{current}$. Consider 2.3, where we have two graphs G and H and the algorithm is already in the middle of its execution ($S_{current} \neq \{\}$). The current vertex matchess are $S_{current} = \{\langle A,6\rangle, \langle C,1\rangle, \langle D,5\rangle\}$. In the next step of the algorithm, we need to find a new pair of unmatched vertices that would not break the MCS definition if added to $S_{current}$.

To do so, in 2.1 we assign a binary bidomain label to each unmatched vertex based on its constraints. For example, vertex $B \in G$ has the label 010 because it is connected to C, but it is not connected to A or D. Vertex $2 \in H$ has the same label 010 because it has the same connections, considering the matches of $S_{current}$. For this reason, B and 2 are both in bidomain 010 and $\langle B,2\rangle$ is a valid candidate that could be added to $S_{current}$. On the other side, vertex G has the label 010 and vertex 3 has the label 000, so they are not in the same bidomain and cannot be added to $S_{current}$. This is because in the new match set, the vertex G would have one neighbor (C) and 3 would have no neighbors, which is not allowed by the isomorphism requirement of the MCS problem.

For future reference, we call $\beta = \{\beta_{left}, \beta_{right}\}$ a generic bidomain, where β_{left}

Table 2.1: Assignment of bidomain labels in McSplit

	Graph G				Graph H			
Vertex	**A**	**C**	**D**	**Bidomain**	**Vertex**	**6**	**1**	**5**	**Bidomain**
B	0	1	0	010	**b**	0	1	0	010
G	0	1	0	010	**c**	0	0	0	000
H	0	1	1	011	**d**	0	1	1	011
E	1	0	1	101	**g**	1	1	1	111
F	0	0	0	000					

and β_{right} are respectively the sets of vertices in G and H that have the same bidomain label. In the example of 2.3, for bidomain 010 we have $\beta_{left} = \{B, G\}$ and $\beta_{right} = \{2\}$. A bidomain β implicitly defines a set of candidate pairs, which are all the possible combinations of vertices in β_{left} and β_{right}. In the example, the candidate pairs for bidomain 010 are $\{\langle B{,}2\rangle, \langle G{,}2\rangle\}$. We call $\mathbb{B}$ the collection of all bidomains.

In line 15, Algorithm 1 selects the smallest of the bidomains containing at least one candidate pair $\langle v, w\rangle$, selects one of the vertices $v \in \beta_{left}$, then it iterates over all the vertices $w \in \beta_{right}$ in the bidomain and recursively calls the algorithm with the updated graphs G' and H'. The updated graphs are obtained by removing the vertices v and w and their adjacent edges from G and H respectively. The vertices are selected in line 16 and line 18 in order, from the node with the highest degree to the one with the smallest. After selecting each new pair (line 9), the algorithm updates the bidomains (line 21) to incorporate the constraints imposed by the newly matched vertices. This is the most cumbersome operation of the algorithm, with a worst-case time complexity of $O((|G|+|H|)\log(|G|+|H|))$. Once the iteration is finished, the current search branch is considered exhausted and the algorithm makes a new recursive call to itself to select the next vertex v and repeat the process.

The algorithm as described until now is performing an exhaustive sweep of the search space, but the true optimization of McSplit lies in its pruning strategy. At line 11 it computes the bound of the current solution. The bound is a limit on the maximum size of the solution that can be generated by the current branch without breaking the MCS definition. Given a generic bidomain β, the maximum number

of matches it can produce is $Bound(\beta) = min(\beta_{left}, \beta_{right})$. Therefore, the bound of the current branch is the sum of the current solution size and all the bounds of all the bidomains.

$$Bound = |S_{current}| + \sum_{\beta \in \mathbb{B}} Bound(\beta) \quad (2.2)$$

If the total bound is smaller than the best solution S, the current branch cannot lead to a better solution, so the algorithm can prune that branch without compromising the quality of the final result.

2.3.3 McSplitSO and McSplitSD

Due to its simplicity, the McSplit algorithm has been extended multiple times. The first small contribution comes from Trimble [23] who observed that the order of the graphs G and H could influence the performance of the algorithm.

```
1 Function McSplitSO(G, H)
2     if |V_G| < |V_H| then
3         S = McSplit(G, H)
4     else
5         S = McSplit(H, G)
6     end
7     return S
8
```

Algorithm 2: The McSplitSO algorithm. The input graphs are swapped based on their vertex count.

McSplitSO compares the sizes of the two input graphs and changes their order accordingly. The optimization exploits the intrinsic asymmetry in the vertex selection and the bound-based pruning strategy in McSplit. Through practical experimentation, Trimble has observed that arranging the smaller graph as the first input reduces the size of the bounds at the lower levels of the search tree. Consequently, this arrangement enhances the pruning effectiveness.

On the other hand, McSplitSD (SD standing for "Swapping of Density") takes a different approach by considering the density of the graphs. The density of a graph is calculated using the formula

$$d(G) = \frac{2|E_G|}{|V_G|(|V_G| - 1)} \quad (2.3)$$

In Algorithm 3, the densities of the graphs are compared using a metric called "density extremeness" (line 2). This metric assigns a higher value to a graph with a

density close to 0 (indicating an independent set) or 1 (indicating a clique), and a lower value to a graph with a density in the middle range. Based on this metric, McSplitSD swaps the order of the graphs and places the less extreme graph first. As with McSplitSO, it has been found through experimental observations that this arrangement improves the pruning performance of the algorithm.

```
1 Function McSplitSD(G, H)
2     if |1/2 − d(G)| < |1/2 − d(H)| then
3         S = McSplit(G, H)
4     else
5         S = McSplit(H, G)
6     end
7     return S
8
```

Algorithm 3: The McSplitSD algorithm. The input graphs are swapped based on their density.

Trimble also proposes an adaptive version of these optimizations, named McSplit2S. However, the resulting modification is not easily applicable to the other McSplit variants that will be explored in the next sections, so it will not be considered.

2.4 Reinforcement Learning (RL)

The next steps forward in the evolution of McSplit are based on Reinforcement Learning. We take a brief pause to explain better this technique, as it will hold considerable importance thorough this thesis.

2.4.1 Theoretical overview

Reinforcement Learning (RL) is a distinct paradigm within the field of Machine Learning (ML), setting itself apart from traditional supervised and unsupervised learning approaches. While supervised learning relies on labeled examples and unsupervised learning seeks to find patterns in unlabeled data, RL takes a unique approach to the learning process through trial and error. In RL, an agent interacts with an environment by observing the current state, performing actions, and receiving feedback on those actions in the form of a new state and a reward score that represents their goodness. The objective of RL is to develop algorithms that enable the agent to learn an optimal policy (a mapping from states to actions) that maximizes the cumulative reward over time. This focus on cumulative rewards

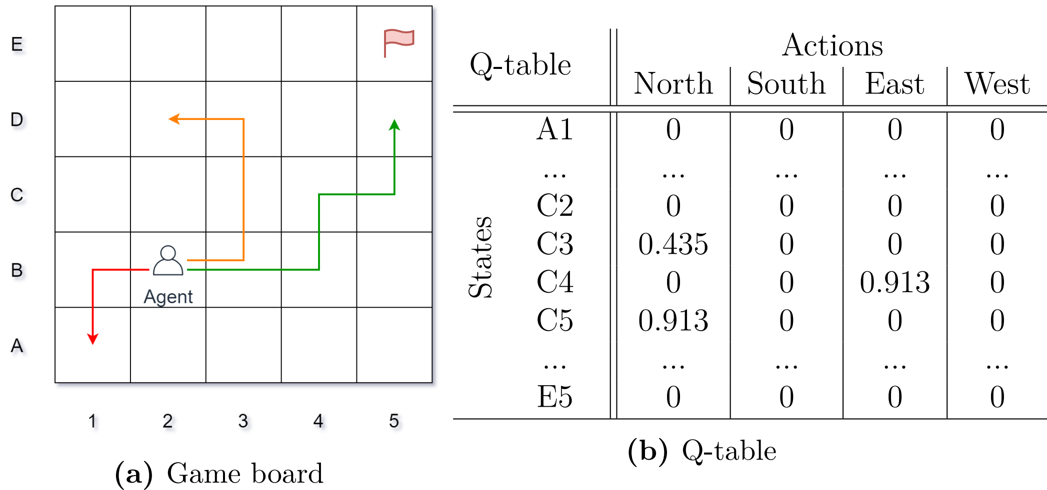
distinguishes RL as a powerful tool for addressing sequential decision-making challenges in dynamic environments.

At the core of RL is the notion of the Markov Decision Process (MDP) [24], which provides a mathematical framework to model the interaction between the agent and the environment. A MDP consists of a set of states, a set of actions, transition probabilities that define the dynamics of the environment, and reward functions that assign numerical values to the agent's actions and states. RL algorithms utilize the MDP framework to learn the optimal policy by iteratively exploring the environment, updating the agent's policy based on the observed rewards, and improving its decision-making abilities over time.

Reinforcement learning encompasses various approaches, with one prominent method being Q-learning. Q-learning is a value-based RL algorithm that aims to learn the optimal action-value function, known as the Q-function. The Q-function follows the optimal policy and outputs the expected cumulative reward for taking a particular action in a given state. By iteratively updating Q-values based on observed rewards and applying an exploration-exploitation trade-off, Q-learning converges to the optimal Q-function and, consequently, the optimal policy [25].

The power of RL lies in the fact that it does not require large labeled datasets, which might be difficult, costly or impossible to obtain, nor it needs a model of the environment. The only requirement is a feedback function that approximates the environmental response to the agent's actions.

2.4.2 An example of Q-learning



(a) Game board

Q-table		Actions			
		North	South	East	West
States	A1	0	0	0	0
	...	...	...	...	...
	C2	0	0	0	0
	C3	0.435	0	0	0
	C4	0	0	0.913	0
	C5	0.913	0	0	0
	...	...	...	...	...
	E5	0	0	0	0

(b) Q-table

Figure 2.4: Example of Q-learning

2.4 illustrates a popular example of Q-learning applied to game theory.

In this example, the agent is a robot that must navigate a 5x5 grid world to reach a goal in cell E5. The agent can move in four directions (North, South, East, West) and it can travel a certain distance until a termination condition is reached. In this case, the position on the board represents the state of the agent, while the four directions are the four possible actions. The movement of the agent is a series of steps, identified by the position (*State*) and the action that the agent takes (*action*). The reward function assigns a value of 1 when the goal is reached and 0 otherwise.

The Q-table represents the agent's Q-function, which is initialized with zeros for all state-action pairs. Each cell of the Q-table is referred to as a Q-value. The agent begins in state B2 and selects an action (e.g., East) based on the current Q-values. The agent then transitions to the next state (e.g., B3) and receives a reward of 0, as it did not reach the goal. Using the Bellman equation [26], the agent updates the Q-value for the state-action pair (B2, East) based on the observed reward and the maximum Q-value for the next state (B3).

Throughout the iteration process, the agent learns that cells D5 and E4 have a maximum Q-Value of 1 since they can reach the goal in one action. Cells C5, D4 and E3 cannot reach the goal directly, but they can reach a state with a Q-value of 1, therefore they get a relatively high reward. The agent continues to explore the environment, updating Q-values based on observed rewards, until it converges to an optimal policy. In this example, one of such optimal policies is to move East from state C3 and then North from B5 to reach the goal state E5.

2.4.3 McSplitRL

The original McSplit algorithm uses a degree-based heuristic to determine the order in which the vertices are selected during the recursive search (lines 16, 18 in Algorithm 1). While the order of vertex selection may not be significant in a normal complete search, in McSplit it plays a vital role in optimizing the search process. By visiting the best nodes first, the algorithm can quickly identify and prune unpromising branches, leading to faster convergence and improved efficiency. This is particularly important considering that McSplit tends to have long execution times, especially for medium to large-sized graphs, therefore, to mitigate this issue, a timeout is often employed. If we are able to select the best vertices first, we might find better solutions within the given timeout.

Liu et al. [27] propose a McSplit variant that uses Reinforcement Learning, called McSplitRL. McSplitRL saves two tables of rewards, which stores the individual rewards of all the vertices in G and H. Given a vertex $v \in G$, we call its score $S_G(v)$. At the beginning of the algorithm, these rewards are initialized to zero. During the recursion, when a new pair of vertices (v, w) is matched and added to the local solution, McSplitRL computes a reward $R(v, w)$ which is proportional to

the reduction in the bound achieved by the match.

$$R(v, w) = \sum_{\beta' \in \mathbb{B}'} min(|\beta'_{left}|, |\beta'_{right}|) - \sum_{\beta \in \mathbb{B}} min(|\beta_{left}|, |\beta_{right}|) \quad (2.4)$$

We call $\mathbb{B}'$ the set of bidomains before the match, and $\mathbb{B}$ the set of bidomains after the match. The reward $R(v, w)$ is then added to the accumulated rewards of both vertices, v and w.

$$\begin{aligned} S_G(v) &= S_G(v) + R(v, w) \\ S_H(w) &= S_H(w) + R(v, w) \end{aligned} \quad (2.5)$$

During vertex selection, McSplitRL chooses the node with the higher accumulated reward, as it indicates a higher potential to reduce the local search space. The selected branch will be faster to explore, and eventually prune if it is not promising, thus increasing the number of visited branches within the given timeout.

2.4.4 McSplitLL

While McSplitRL presents improvements over the original McSplit algorithm, it also faces a challenge regarding the staleness of rewards during the progression of the recursion. As the search moves into different areas of the search tree, the rewards computed earlier may no longer accurately reflect the current state of the graph. Most commonly, a vertex $w \in H$ might be an optimal candidate for selection in a certain branch of the search tree, but not in another.

This is one of the main issues that McSplitLL [28] aims to address. McSplitLL is a variant of McSplit that introduces two new independent improvements: *Long-Short Memory Branching Heuristic* and *Leaf Vertex Union Match Strategy*.

Long-Short Memory (LSM) Branching Heuristic To address the reward staleness issue, McSplitLL changes the reward structure of the Q-learning infrastructure. While vertices v still have an individual reward $S_G(v)$, the reward of w vertices is now dependent on the currently selected vertex v, to which w is matched. We call this pair reward $S_p(v, w)$, and it behaves as a reward on the actual matches. While in McSplitRL the rewards had a memory occupation complexity of $O(|G| + |H|)$, in McSplitLL it expands to $O(|G||H|)$. The reward update policy is trivially changed from 2.5 to 2.6.

$$\begin{aligned} S_G(v) &= S_G(v) + R(v, w) \\ S_p(v, w) &= S_p(v, w) + R(v, w) \end{aligned} \quad (2.6)$$

Under Q-learning terminology, the selected vertex v represents the state on which is applied the action of matching a vertex w. The pair reward $S_p(v, w)$ is the Q-value of the state-action pair (v, w).

On these two sets of rewards, McSplitLL applies an independent decay mechanism, based on two threshold values $t_v < t_p$. After the reward update, if $S_G(v) > t_v$, then all rewards $S_G(\cdot)$ are decayed by a factor α (by default $\alpha = \frac{1}{2}$). The same process applies to all $S_p(\cdot, \cdot)$ scores, when $S_p(v, w) > t_p$. The LSM mechanism uses two different thresholds because the pair rewards are more specific and naturally more precise than the individual rewards, therefore they are valid for a longer period of time. On the other hand, the vertices v are selected more dynamically and their rewards tend to become stale faster.

Leaf Vertex Union Match (LUM) Strategy The second improvement leverages the concept of leaf vertices in an undirected graph. A leaf vertex v_l is a vertex with degree 1, meaning that it is connected to only one other parent vertex v_p. If $v_p \in G$ and $w_p \in H$ are matched, then their connected leaves v_l and w_l are necessarily in the same bidomain and can be matched automatically.

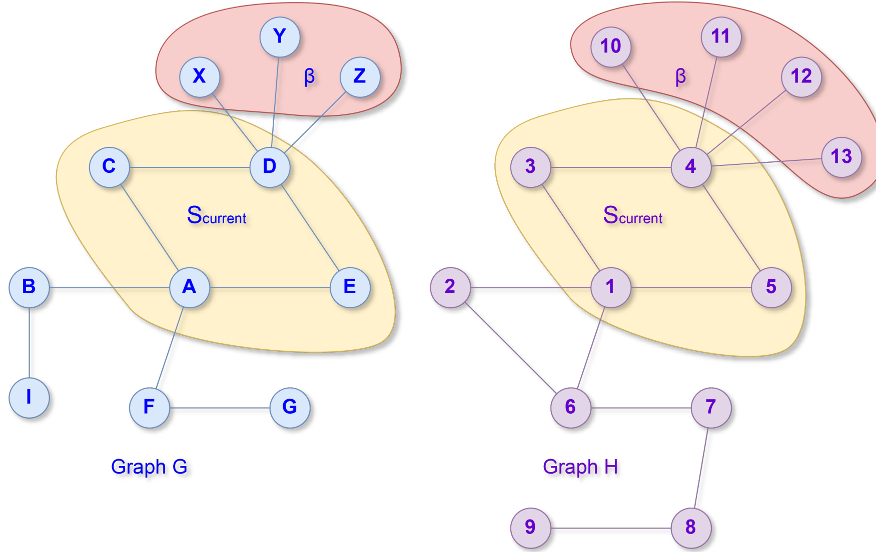


Figure 2.5: Leaf Vertex Union Match. If $\langle D,4 \rangle \in S_{current}$, the leaf vertices in bidomain β can be automatically matched pair-by-pair, in any order. Since the number of leaves is different on the two sides of the bidomain, one leaf will remain unmatched.

If two matched vertices $\langle v_p, w_p \rangle$ are connected to multiple leaves (2.5), then it can be trivially proven that all leaves can be immediately matched pair-by-pair regardless of the order (for instance, $\langle X,10 \rangle$, $\langle Y,11 \rangle$, $\langle Z,12 \rangle$). This occurs

because the leaves are not directly connected to each other, so their match cannot influence any other node. If the number of leaves is different on the two sides of the bidomain, some leaves will remain unmatched. This improvement, while based on a simple observation, can lead to significant performance improvements, as it reduces the number of recursive calls and the number of vertices to be considered in the branching heuristic. Moreover, this mechanism does not affect the optimality of the algorithm.

2.5 Machine Learning meets graphs: GNNs

In recent years, Machine Learning (ML) has emerged as a dominant field in computer science, and it is now an essential tool for many research projects in all domains. However, traditional linear neural networks or other models that operate on fixed-dimensional input vectors are not well suited for handling graph-structured data. Graphs are inherently different from regular tabular data or images because they capture complex relationships and dependencies among a variable number of elements.

2.5.1 Multi-Layer Perceptrons (MLPs)

A Neural Network (NN) is a basic category of ML architecture that performs a non-linear transformation of the input features by means of an activation function. It is a fundamental ML model used for tasks such as binary classification or non-linear regression.

Perceptron A NN consists of a single layer of artificial neurons, called perceptrons, which take a set of input features X, multiply them by corresponding weights W, and sums them up. Each perceptron can also have a bias b that is added to the total. The weighted sum is then passed through an activation function $f(\cdot)$, for example a step function, to produce the output y. The activation function makes the perceptron a non-linear function, allowing it to process and learn more complex information. Specifically, the step function assigns a value of 1 if the weighted sum exceeds a certain threshold and 0 otherwise, but there is a wide range of other activation functions that can be used.

Mathematically, the output of a perceptron can be represented as follows:

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{2.7}$$

The weights and the bias of the perceptron are initially assigned random values and are updated during the training process using algorithms such as gradient

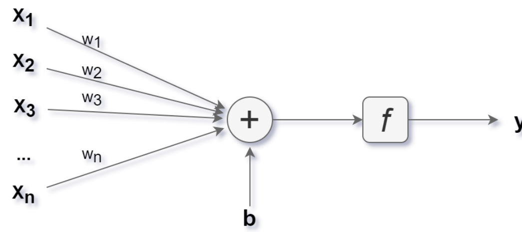


Figure 2.6: A perceptron with n inputs

descent, to minimize the error in the network's predictions. This error, or loss, is a function that computes a distance metric between the output of the perceptron and a given label $\hat{y}_i$, which represents the desired outcome. This dependency on labels makes the NN a supervised model. A popular loss function is the Mean Squared Error (MSE), defined as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.8}$$

By minimizing the loss, the perceptron is able to learn the optimal set of parameters that best produce the desired outcome on the input data. This knowledge can then be used to make predictions on new data.

Multi-Layer Perceptron A Multi-Layer Perceptron (MLP) is a network composed of multiple layers of perceptrons stacked on top of each other. The output of each layer of neurons is passed as input to the next, and the final output is computed by the last layer. The output of each perceptron is passed through an activation function before being passed as input to the next layer. In these more complex structures, the activation function is typically applied element-wise to the output of each perceptron in a layer, but each layer can have a different activation function.

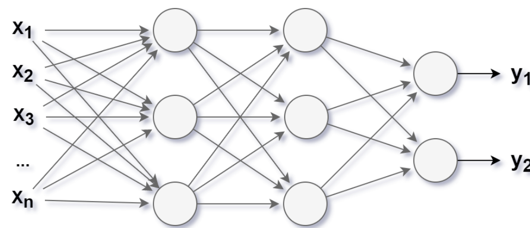


Figure 2.7: A multi-layer perceptron (MLP) with three layers. Each circle represents a perceptron with a bias (not shown), and each line represents a connection between neurons with an individual weight.

Two popular activation functions used in MLPs are the sigmoid or the rectified linear unit (ReLU). ReLU is defined as

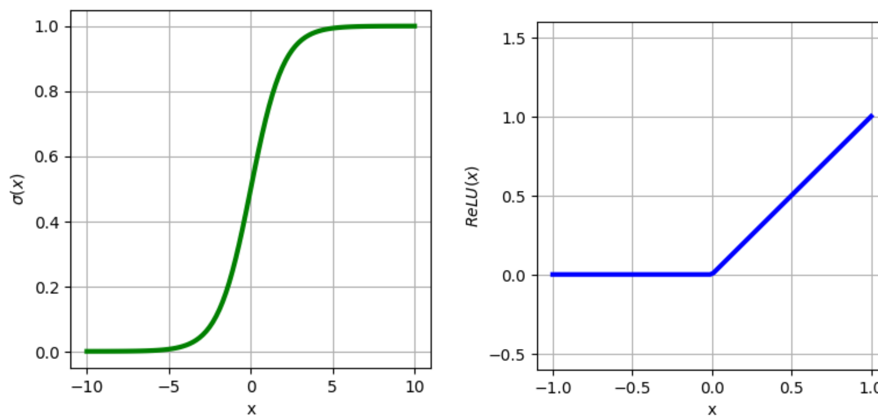
$$ReLU(x) = \max(0, x) \tag{2.9}$$

and is a non-linear function that is zero for negative values and linear for positive values (Figure 2.8b). It is a commonly used activation function in neural networks because it is simple to compute and its only operation is to discard the negative values, leaving the positive ones unaltered. However, the ReLU function is not differentiable at $x = 0$, and it has a null gradient for $x < 0$, which can cause problems during the training process, so sometimes the LeakyReLU variant is used.

The sigmoid function is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.10}$$

and is an S-shaped curve that maps the input to a value between 0 and 1 (Figure 2.8a). It is commonly used in the output layer of a neural network to rescale the final scores to values between 0 and 1, but it is not recommended for hidden layers because it is prone to the vanishing gradient problem. The vanishing gradient problem occurs when the gradient of the activation function becomes very small (as in the case of the sigmoid when $x \to \pm\infty$), causing the weights to be updated very slowly and the learning process to possibly stagnate.



(a) Sigmoid Function

(b) ReLU Function

Figure 2.8: Popular activation functions

2.5.2 Graph Neural Networks (GNNs)

MLPs assume that the input data is in a fixed-dimensional format, where the order or connectivity between elements is irrelevant or unknown. This assumption fails to capture the structural information present in graphs, such as node connections and neighborhood relationships. Additionally, traditional models have a predefined number of perceptrons in the first layer, which must always be equal to the number of input features. Consequently, they lack the ability to handle varying-sized inputs, which is a fundamental characteristic of graph data.

To overcome these limitations, Graph Neural Networks (GNNs) have gained attention as a specialized class of models designed explicitly for graph-structured data. GNNs leverage the inherent connectivity and relational information present in graphs to learn and make predictions. By performing message passing and aggregation operations across nodes, GNNs can capture the complex dependencies and structural patterns within graphs.

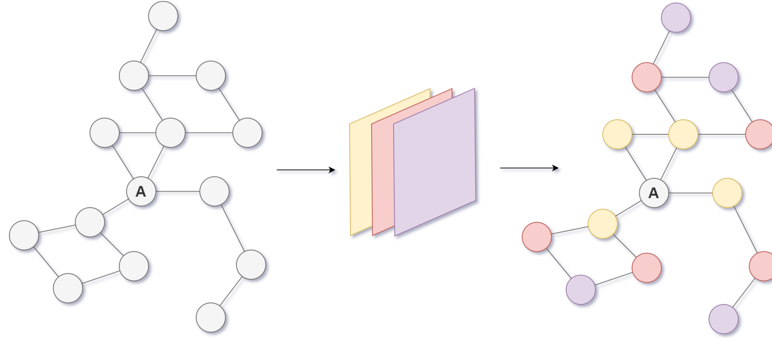


Figure 2.9: A graph convolutional network (GCN) with three layers. Considering node A, in each layer the node aggregates information from its neighbors, which themselves store information about their neighbors, therefore expanding its knowledge to all nodes within a 3-hop radius.

Graph Convolutional Networks (GCNs)

The most common GNN architecture is the Graph Convolutional Network (GCN). In each convolutional layer, each node in the graph collects information from its neighboring nodes (message passing) and updates its own representation based on this information. This process is iteratively performed across multiple layers, allowing nodes to gather information from distant parts of the graph. By combining local and global information, GNNs strive to capture the structural patterns and properties in the graph. By adjusting the number of layers of the GCN, the model can learn to incorporate increasingly complex information and identify patterns of

a desired size.

The learning process of a GCN happens in each layer after the message passing operation. Once a node has collected the information about its neighbors, it stores it in a hidden state vector. While the number of vertices is variable from graph to graph, the dimensionality of the node vector in each layer is defined a priori. Therefore, the vector can then be passed through a MLP, which outputs a learned representation of the node, which is then passed to the next layer. The output of the final layer is the node embedding, which is a vector representation of the node and its neighborhood that captures its structural properties and can be used for downstream tasks.

Graph Attention Networks (GATs)

Another widely recognized GNN architecture is the Graph Attention Network (GAT) [29]. GATs share similarities with GCNs in their ability to aggregate information from neighboring nodes. However, a key distinction is that GATs employ K attention heads to learn a weighted combination of neighbor representations. This mechanism allows GATs to incorporate a cognitive attention mechanism, enabling the model to selectively emphasize relevant neighbors while disregarding irrelevant ones. The output of each GAT layer is represented by:

$$\mathbf{y} = \bigcup_{k=1}^{K} f(\alpha_k \mathbf{W}\mathbf{X}) \tag{2.11}$$

where $\mathbf{X}$ is the input node feature matrix, collecting the vector features of all vertices, $\mathbf{W}$ is the weight matrix, f denotes a generic activation function, and α_k corresponds to the attention coefficient associated with the k-th attention head. Each attention coefficient α_k is a trainable parameter that is learned during the model training process. Consequently, incorporating attention heads in GATs introduces additional trainable parameters, slightly augmenting the overall size of the model.

Chapter 3

Algorithmic Optimizations and Heuristics

In Chapter 2 we introduced the McSplit algorithm and we discussed the improvements made by McSplitRL and McSplitLL, which were achieved using a smarter node selection heuristic based on Reinforcement Learning. While these changes improved the performance of McSplit, they also introduced some additional overhead. McSplitRL needs to spend a significant amount of clock cycles keeping the Q-table of rewards up to date, and in McSplitLL the overhead is even greater due to the increased size of the Q-table. This implies that these algorithms try to have a smarter vertex selection policy to increase the pruning rate and home in on the optimal solution faster, at the cost of having overall slower iterations. We call these dynamic heuristics, as the vertex selection policy is updated on-the-fly as the algorithm runs.

In this chapter, we will first build different implementations of McSplitDAL, the state-of-the-art solver at the time of the research project, in search of possible algorithmic optimizations. For future reference, we will refer to the family of the main McSplit variants (McSplit, McSplitLL, and McSplitDAL) as McSplitX.

We will then try to improve the performance of McSplitX using what we will call static heuristics. The considered heuristics will be PageRank (PR), Betweenness Centrality (BC), Closeness Centrality (CC), Katz Centrality* (KC*), and Local Clustering Coefficient (LCC). We refer to the family of McSplitX variants with static heuristics as McSplitX+. These heuristics will produce scores based on the properties of the graph, which will not change during the execution of the algorithm. We will then compare the performance of these different implementations, and try to understand the impact of the different heuristics on the sizes of the solution produced by the algorithms.

3.1 McSplitDAL

3.1.1 Overview

McSplitDAL [30] was the state-of-the-art algorithm for solving the MCS problem during the research project. As in the case of McSplitLL, it brought 2 contributions to the McSplit algorithm: a modified reward function called Domain Action Learning (DAL), and a hybrid learning policy that combines DAL with the original RL policy.

Domain Action Learning

In McSplitRL the reward was uniquely based on the bound reduction achieved by the selected match. The authors of McSplitDAL argue that this reward function can be integrated with a new component that takes into account the reduction in complexity of the updated bidomains.

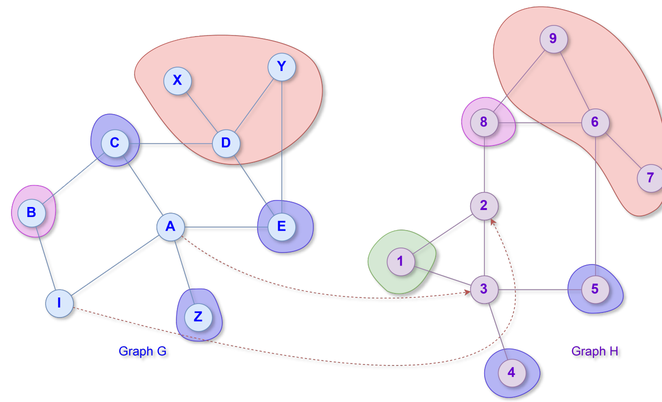
Consider 3.1. In Figure 3.1a we have a graph with four bidomains. When the pair $\langle B{,}1\rangle$ is matched (Figure 3.1b), the total number of bidomains remains unchanged. However, when the pair $\langle D{,}6\rangle$ is matched (Figure 3.1c), the number of bidomains is increased to six. Since the number of unmatched vertices after the match is necessarily the same, it is intuitive that the bidomains will have to be smaller on average. This effect is clearly visible in Figure 3.1c. Having a higher number of smaller bidomains increases the likelihood of finding leaves, which can be easily matched using the LUM strategy of McSplitLL. Furthermore, smaller bidomains might also be immediately pruned, as they have a smaller bound.

To reward matches that increase the number of bidomains, and therefore decrease their size, the authors of McSplitDAL propose the following modification to the reward function:

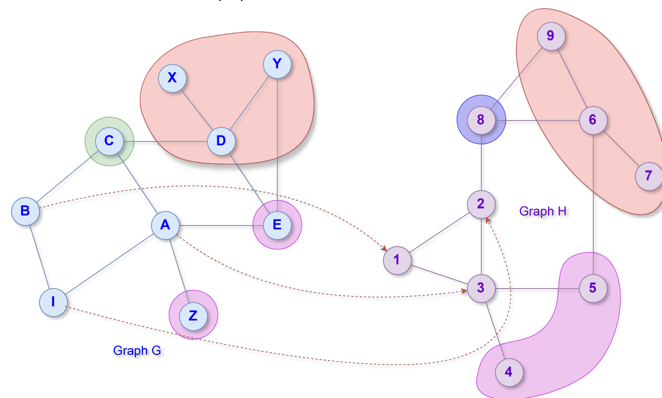
$$R(v,w) = \sum_{\beta' \in \mathbb{B}'} min(|\beta'_{left}|, |\beta'_{right}|) - \sum_{\beta \in \mathbb{B}} min(|\beta_{left}|, |\beta_{right}|) + |\mathbb{B}| \tag{3.1}$$

Hybrid learning policy

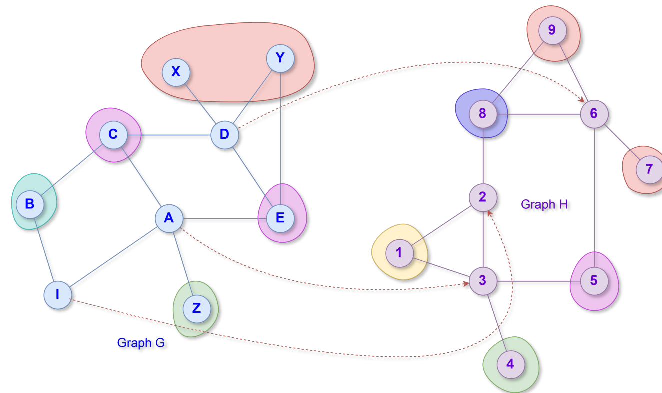
The hybrid learning policy of McSplitDAL is a combination of the original RL policy and the DAL policy. The authors note that the RL and DAL agents are prone to fixation on a local minimum. A workaround to this problem is to switch between the RL policy and the DAL policy every number of iterations $MaxNbApp$. The switching is orchestrated by a counter $NbApp$ that is reset to zero when a policy switch occurs ($NbApp > MaxNpApp$) or when a new best solution is found ($|S_{current}| > |S|$). The rationale for this mechanism is that the two agents are



(a) Before matching



(b) After matching ⟨B,1⟩



(c) After matching ⟨D,6⟩

Figure 3.1: Example of the DAL reward function.

different enough to help the algorithm to escape local minimums, but they are still both good enough to find a satisfactory solution.

3.1.2 Our implementations

The authors of McSplitDAL did not provide an official implementation of the algorithm. Therefore, we implemented our own version of the algorithm in C++, and we tested some small modifications to the algorithm on the way. Here are reported some of the most notable variants we produced.

Joint vs Isolated Q-tables

The first modification we tested was the use of joint Q-tables versus isolated Q-tables. McSplitDAL uses an alternation of the RL policy and the DAL policy, which define different reward update functions. In the first version, McSplitDAL Joint, there is a common Q-value pool for both policies, and only the reward update function is routinely changed. In the second version, McSplitDAL Isolated, there are two separate Q-value pools, one for each policy. The update function is computed simultaneously for both policies, but only the Q-value pool of the current policy is interrogated during a McSplit branching event.

McSplit Isolation necessarily requires double the memory capacity and the computational power to store and update the Q-value pools. However, it also allows the two policies to learn independently of each other. This allows the policies to differentiate more, and therefore to escape local minimums more easily.

Initialization of Q-tables

Traditional RL approaches rely on a starting initialization to zero of the Q-values, which are then updated over the course of numerous iterations. In McSplitDAL (or in general in all derivatives of McSplitRL) the algorithm is executed only once, but the rewards are still initialized to zero. At the start of the recursion, McSplit has to decide which candidate match to select, and since all the Q-values are initialized to zero, the algorithm will select as a tie-breaker the vertex with the highest number of incident edges, as provided by the node-degree heuristic. Therefore, in the first stages of the recursion, the algorithm will act similarly to the original McSplit while the Q-table fills up.

However, the first branching decisions are arguably the most important ones: each branch might require a long time to be explored, and therefore the algorithm might not have enough time to explore all of them. Consequently, it is important to make sure that the first branching decisions are the best ones. In the current state, if during a recursion the algorithm has to decide between two vertices, one already encountered and one new, it will always select the already encountered vertex, as it will have a non-zero Q-value. This is not necessarily the best choice, as the new vertex might have a higher degree, and it might be a better candidate.

To solve this problem, we test a different McSplitDAL variant, where the Q-tables are initialized with the scores provided by the heuristic (i.e., the degree of the node). This way we assign a stronger initial weight to the heuristic during the first recursions, which will then slowly decade as the rewards follow their update-decay cycle.

McSplitRL+DAL vs McSplitLL+DAL

McSplitDAL alternates the RL policy and the DAL policy to have two different search agents that can collaborate to escape local minima. However, the RL policy is less performant than other search agents, such as McSplitLL. Therefore, we test a variant of McSplitDAL where the RL policy is replaced by McSplitLL. For clarity, we call this variant McSplitLL+DAL, differently from the original McSplitDAL, which we temporarily call McSplitRL+DAL.

The two implementations represent a trade-off: McSplitLL+DAL has two good agents, but they are both similar and therefore they might not be able to escape local minima. McSplitRL+DAL has two different agents, but one of them is less performant than the other, and therefore it might not be able to find good solutions. We will test the two variants to see which one performs better in practice.

3.2 Static Heuristics

As discussed in Algorithm 1 (Section 2.3), McSplit uses a higher-degree-first heuristic to select the next vertex to branch on. Since the degree of the nodes is constant, it can be computed once at the start of the algorithm. An efficient implementation is shown in Algorithm 4. The algorithm first computes the degree of each vertex (lines 2-3) as a numerical score, then sorts the vertices of G and H in descending order of these scores (lines 4-5). In our implementation, the graphs are represented through an adjacency list to reduce memory consumption, but the approach is valid with other data structures as well. The actual recursive McSplit function is then called on the sorted graphs.

```
1 Function McSplit(G, H)
2     g_order = computeHeuristic(G)
3     h_order = computeHeuristic(H)
4     G~ = sort(G, g_order)
5     H~ = sort(H, h_order)
6     S = mcs(G~, H~, {})
7     return S
8
```

1 **Function** `McSplit`(G, H)
2 $g_{order} = computeHeuristic(G)$
3 $h_{order} = computeHeuristic(H)$
4 $\widetilde{G} = sort(G, g_{order})$
5 $\widetilde{H} = sort(H, h_{order})$
6 $S =$ `mcs`$(\widetilde{G}, \widetilde{H}, \{\})$
7 **return** S
8

Algorithm 4: Implementation of McSplit that sorts the vertices of G and H according to a heuristic before starting the search.

Inside the `mcs()` function of the basic McSplit algorithm, the vertices are selected in the order they appear in the sorted graphs, thus respecting the heuristic without any explicit and expensive calculation during the recursion. In the case of the more complex RL-based McSplit variants, the selection functions use the Q-table rewards to select the vertices, but they still rely on the sort order as a tie-breaker (Algorithm 5). In Chapter 6 we will confirm through experimental observations that the heuristic still holds a significant impact on the performance of the RL-based algorithms.

McSplit uses the *degree* heuristic in the hope that highly connected vertices are more likely to be part of the solution. However, this metric fails to account for the properties of the neighborhood of the node and is therefore extremely limited. This simplicity makes the heuristic particularly fast to compute, but this is not an important property, as the sort order is computed only once at the start of the algorithm. Moreover, the degree is by its nature a low-variance integer number, so most nodes will have the same score and ties will be frequent.

Next, we propose alternative heuristics that should have the following properties:

- They should produce high-variance floating-point scores, to decrease the

```
1  Function SelectVertexV(G, β)
2      vertices = getBidomainVertices(G, β)
3      max_reward = 0
4      best_vertex = null
5      for v ∈ vertices do
6          reward = getRLreward(v)
7          if reward > max_reward then
8              max_reward = reward
9              best_vertex = v
10         end
11     end
12     return best
13
```

Algorithm 5: `selectVertexV` function that selects the next vertex v to branch on, using RL rewards. The static heuristic is implicitly used as a tie-breaker, since if multiple nodes have the maximum reward, only the first in the heuristic sort order is selected. The selection function for w is similar.

probability of ties.

- They should be relatively fast to compute so that the overhead of calculating them is negligible compared to the time spent in the recursive calls.

Regarding the second requirement, we need to consider that the graphs need to be sorted on the computed scores, with a worst-case complexity of $O(|V|^2 \log |V|)$, for a graph $G(V, E)$. Our main goal is to keep the complexity of the heuristics lower or comparable to this threshold. In practice, the heuristic computation might also be performed in parallel, if the selected algorithm allows it.

Next, we will examine five alternative heuristics designed for the McSplit algorithm. These new candidate sort orders have been chosen based on their ability to prioritize the vertices in different ways, with the objective of establishing a more effective best-first node selection policy.

3.2.1 PageRank (PR)

PageRank [31] is a crucial algorithm that revolutionized web search and information retrieval. Its implementation paved the way for the emergence of Google, as it successfully addressed the limitations of previous ranking methods. Developed by Larry Page and Sergey Brin at Stanford University, PageRank aims to quantify the importance and relevance of web pages based on their inbound link structure. By assigning a numerical score to each page, PageRank provides a reliable measure of its authority and popularity within the vast web ecosystem. This algorithm evaluates both the number and quality of incoming links, considering links from high-ranking pages as more valuable. Consequently, PageRank not only enhances search engine rankings but also enables the identification of influential web pages.

As the world wide web can be represented as a directed graph, the importance metric computed by PageRank can be directly applied to the nodes of the graphs in the MCS problem. The intuition is that a node is important if it is pointed to by other important nodes. Since in our research we are focusing on undirected graphs, in Algorithm 6 we report an undirected version of PageRank.

The algorithm computes a stochastic matrix G_s from the input graph G (lines 5), which represents the flow of information in each edge. If we pretend that each node produces one unit of information, and that this information proportionally flows in all incident edges to the adjacent nodes, then the stochastic matrix G_s represents the relative amount of information that a node u will send to a node v.

G_s represents a Markov chain, where each node is a state and the edges are the transitions between states. The objective of PageRank is to compute the stationary distribution of the Markov chain, which is the probability that a random walk on the graph will be in each node. The stationary distribution is computed by iteratively multiplying the stochastic matrix G_s with a vector p containing the probability

```
1  DF ← 0.85
2  ϵ ← 0.00001
3
4  Function PageRank(G)
5      G_s ← computeStochasticMatrix(G)
6      G_{s,t} ← G_s^T
7      p ← [1/|G|] ∈ ℝ^{|G|}
8      while error > ϵ do
9          ranks ← [0] ∈ ℝ^{|G|}
10         ranks ← G_{s,t} · p
11         ranks ← DF × ranks + (1−DF)/|G|
12         errors_{local} = abs(ranks − p)
13         error ← errors_{local} · ([1] ∈ ℝ^{|G|})
14         p ← ranks
15     end
16     return p
17
18 Function computeStochasticMatrix(G)
19     G_s ← [0] ∈ ℝ^{|G|,|G|}
20     forall u ∈ G do
21         forall v ∈ G do
22             if deg(u) = 0 then
23                 G_s[u, v] ← 1/|G|
24             else
25                 if (u, v) ∈ E_G then
26                     G_s[u, v] ← 1/deg(u)
27                 end
28             end
29         end
30     end
31     return G_s
32
```

Algorithm 6: PageRank algorithm for undirected graphs.

of being in each node (line 10). In practice, this is achieved by computing a dot product between the transposed matrix G_s, t and p. The algorithm stops when the difference between the probability vectors of two consecutive iterations is less than a threshold ϵ (line 2). This difference is computed as the sum of the differences of the individual node probabilities (line 10 we indicate it as a dot product of the difference vector and the unitary vector).

The damping factor DF (line 1) is a parameter that controls the probability of jumping to a random node instead of following the edges of the graph. This is necessary to avoid the case where a node has no outgoing edges, which would cause the algorithm to get stuck in that node. The damping factor is the main feature that differentiates PageRank from the Eigenvector Centrality metric. The PageRank algorithm is guaranteed to converge to the stationary distribution, and the resulting vector p contains the PageRank score of each node.

The time complexity of the algorithm highly depends on the implementation. Algorithm 6 builds the stochastic matrix G_s in $O(|V|^2 + |E|)$, which is easier to handle, then it iterates k times until convergence in $O(|V|)$. This sacrifices both time and memory $(O(|V|^2))$ for the sake of simplicity during the prototyping phase of the research. If the input graphs become too large, PageRank can be reduced to an overall complexity of $O(k(|V| + |E|))$.

3.2.2 Betweenness Centrality (BC)

Betweenness centrality [32] is a measure of centrality in a graph that considers the shortest paths. Given any pair of vertices in a connected graph, there exists at least one path between those nodes such that the number of edges that the path passes through is minimized. This path is called the shortest path. The betweenness centrality for each vertex is the number of these shortest paths that pass through the vertex. If a pair of vertices has multiple paths composed of the minimum number of edges, only one of them is considered.

Algorithm 7 is an implementation of the Betweenness Centrality that uses the Brandes algorithm [33] to compute the shortest paths between all pairs of nodes in the graph and assigns the centrality scores.

In the algorithm we consider each node $s \in G$ separately, and we compute all shortest paths from that node to any other node $w \in G$ using a BFS approach. We use a queue Q to store the nodes that we still need to visit, and a stack S to store the nodes in the order that we visit them. We also use a vector P to store the predecessors of each node in the shortest paths, and a vector σ to store the number of shortest paths from s to each node $w \in G$. We use a vector d to store the distance from s to each node $w \in G$, and a vector δ to store the centrality of each node $w \in G$.

The core of the algorithm is the computation of δ in line 30 using a backtrack

```
1  Function BetweennessCentrality(G)
2      BC ← [0] ∈ ℝ^|G|
3      for each s ∈ |G| do
4          P ← [] × |G|
5          S ← stack()
6          Q ← queue()
7          σ ← [0] ∈ ℝ^|G|
8          d ← [−1] ∈ ℝ^|G|
9          δ ← [0] ∈ ℝ^|G|
10         σ[s] ← 1
11         d[s] ← 0
12         Q.push_back(s)
13         while |Q| > 0 do
14             v ← Q.pop()
15             S.push(v)
16             for each w ∈ neighborhood(v) do
17                 if d[w] < 0 then
18                     Q.push(w)
19                     d[w] ← d[v] + 1
20                 end
21                 if d[w] = d[v] + 1 then
22                     σ[w] ← σ[w] + σ[v]
23                     P[w].append(v)
24                 end
25             end
26         end
27         while |S| > 0 do
28             w ← S.pop()
29             for each v ∈ P[w] do
30                 δ[v] ← δ[v] + (σ[v]/σ[w]) (1 + δ[w])
31             end
32             if w ≠ s then
33                 BC[w] ← BC[w] + δ[w]
34             end
35         end
36     end
37     return BC
```

Algorithm 7: Betweenness Centrality

approach (achieved by fetching the elements of the stack S in the reverse order of their insertion). At each source s and destination w, δ is equal to the ratio of shortest paths passing through v by the number of shortest paths not passing through v. The δ of each node v is then accumulated in the BC global vector.

The computation of the Betweenness Centrality is considerably more complex than the original *degree* heuristic. In the worst case, it has a time complexity of $O(|V|(|V| + |E|))$, but, if needed, more optimized algorithms are available to further reduce the complexity to $O(k|V|)$, with $k < |E|$ [34]. In practice, this implementation has an advantage over PageRank, as the operations on each vertex s are completely independent. This means that the algorithm can be easily parallelized on multiple threads, achieving a speedup that is essentially linear with the number of threads.

3.2.3 Closeness Centrality (CC)

Closeness centrality [35] is a measure of centrality in a graph based on the average shortest path distance between a node and every other node in the graph. It is defined as the reciprocal of the sum of the shortest path distances $d(s, w)$ from the node s to all other nodes w in the graph. Thus, the more central a node is, the closer it is to all other nodes.

$$CC(s) = \sum_{w \in G \setminus \{s\}} \frac{1}{d(s, w)} \tag{3.2}$$

To get a proper average of the distances, we should normalize the equation by $|G| - 1$, but this is not important for our application.

The algorithm, shown in Algorithm 8 is a straightforward computation of the lengths of the shortest paths $d(s, w)$ using Dijkstra's algorithm, followed by the application of 3.2.

For each node, the code has the same complexity as Dijkstra's algorithm, which is $O(|E| + |V| \log |V|)$ if a Fibonacci heap is used. However, versions optimized for the Closeness Centrality and with a lower complexity are available [36]. Regardless, the final complexity of our implementation is quadratic in $|V|$, thus we expect its computation to introduce a noticeable overhead during the execution of McSplit. As in the case of the Betweenness Centrality, the computation of the Closeness Centrality is completely independent for each node s, and thus the algorithm can be easily parallelized on multiple threads.

3.2.4 Katz Centrality* (KC*)

The Katz Centrality (KC) [37] is a measure of centrality that takes into account the number of paths of length l between a pair of nodes. It is defined as the sum

```
1  Function ClosenessCentrality(G)
2      CC ← [0] ∈ ℝ^|G|
3      for each s ∈ |G| do
4          Visited ← [false] ∈ ℝ^|G|
5          d ← [∞] ∈ ℝ^|G|
6          dt ← [∞] ∈ ℝ^|G|
7          d[s] ← 0
8          dt[s] ← 0
9          for j ← 0 to |G| − 1 do
10             Minimum ← ∞
11             for k ∈ |G| do
12                 if Visited[k] = false then
13                     if Minimum > dt[k] then
14                         Minimum ← dt[k]
15                         u ← k
16                     end
17                 end
18             end
19             dt[u] ← ∞
20             for w ∈ neighborhood(u) do
21                 if Visited[w] = false then
22                     if d[w] > d[u] + 1 then
23                         d[w] ← d[u] + 1
24                         dt[w] ← d[w]
25                     end
26                 end
27             end
28             Visited[u] ← true
29         end
30         CC[s] = 1 / sum(d)
31     end
32     return CC
```

Algorithm 8: Closeness Centrality

of the number of paths of length l between a pair of nodes s and w for all $l \in \mathbb{N}$, weighted by a factor $\alpha \in \mathbb{R}$. Unlike the Betweenness and Closeness Centrality measure that used the shortest paths, the KC uses all paths between a pair of nodes, therefore resembling more PageRank and the similar Eigenvector Centrality. However, we used a slightly modified version of KC that we call Katz Centrality*, which considers only the shortest paths. This enables us to compute the algorithm using a single BFS traversal, instead of the longer iterative approach used in the original algorithm, thus keeping the heuristic fast to compute in $O(|V| + |E|)$.

```
1  Function KatzCentrality*(g)
2      α ← 0.5
3      KC ← [0] ∈ ℝ^|G|
4      for s ∈ G do
5          visited ← false ∈ ℝ^|G|
6          score ← [0] ∈ ℝ^|G|
7          score[s] ← 1
8          next_layer = {s}
           // Run BFS
9          while |next_layer| > 0 do
10             current_layer ← next_layer
11             next_layer ← {}
12             for v ∈ current_layer do
13                 for w ∈ neighborhood(v) do
14                     if visited[w] = false then
15                         score[w] ← score[w] + α · score[v]
16                         next_layer = next_layer ∪ {w}
17                     end
18                 end
19             end
20             for w ∈ next_layer do
21                 visited[w] ← true
22             end
23         end
24         KC[s] = sum(score)
25     end
26     return KC
```

Algorithm 9: Katz Centrality*

In Algorithm 9 we can see that KC* computes a metric that extends the traditional degree heuristic. The algorithm computes the shortest paths from a

given vertex s to all other vertices w in the graph. If we call $d(s,w)$ the length of the shortest path from s to w and k the number of paths from s to w with a length of $d(s,w)$, vertex w is assigned a local score given by $k \cdot \alpha^{d(s,w)}$. More in general, if $S_l(s)$ is the set of vertices at minimum distance l from s, the total score of w is given by

$$KC^{*}(s) = \sum_{l=1}^{\infty} |S_l(s)| \cdot \alpha^l \tag{3.3}$$

An alternative interpretation is that the algorithm calculates the l-neighbors of s (i.e., the vertices exactly l edges away from s) and assigns them a score proportional to α^l. By summing the scores of all l-neighbors for each value of l, the algorithm provides an enhanced measure of centrality that captures the influence of neighboring vertices in a comprehensive manner. Following this definition, the *degree* heuristic is a special case of KC* with $\alpha = 1$ computed only for $l = 1$.

3.2.5 Local Clustering Coefficient (LCC)

Most of the previous heuristics try to estimate a value of centrality that is best suited for the MCS problem. On the other hand, Local Clustering Coefficient (LCC) is a metric of how much the nodes tend to cluster together and form a clique. It is defined as the ratio of the number of edges (u,v) connecting the neighboring nodes of a vertex s, and the maximum number of edges that could possibly exist between them.

$$LCC(s) = \frac{2|\{(u,v) \in E : (s,u) \in E \vee (s,v) \in E\}|}{deg_n(s)(deg_n(s)-1)} \tag{3.4}$$

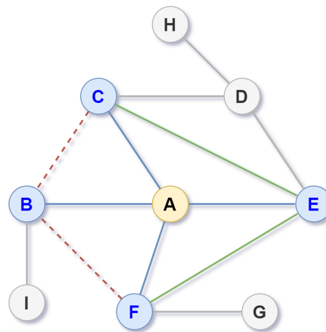


Figure 3.2: Example of the Local Clustering Coefficient (LCC) of a node A with $deg_n(s) = 4$. The LCC is the ratio of the number of edges between the neighboring vertices of A (highlighted in green) and the maximum number of edges that could potentially exist between them (missing edges are shown in red).

We call these edges triangles, because they form a triangle together with the edges (s, u) and (s, v). For brevity, we define $deg_n(s) = |\text{neighborhood}(s)|$, in the knowledge that $deg_n(s) = deg(s)$ if s does not have self-loops (edges (s, s)).

```
1  Function LocalClusteringCoefficient(G)
2      LCC ← [0] ∈ ℝ^|G|
3      for s ∈ G do
4          if deg_n(s) < 2 then
5              LCC[s] ← 0
6          else
7              n_triangles ← 0
8              neighbors ← neighborhood(s)
9              for v ∈ neighbors do
10                 neighbors ← neighbors \ {v}
11                 for w ∈ neighbors do
12                     if (v, w) ∈ E then
13                         n_triangles ← n_triangles + 1
14                     end
15                 end
16             end
17             LCC[s] ← 2n_triangles / (deg_n(s)(deg_n(s)−1))
18         end
19     end
20     return LCC
```

Algorithm 10: Local Clustering Coefficient

The algorithm presented in Algorithm 10 provides a straightforward implementation of 3.4. The algorithm operates by computing the number of triangles present in the neighborhood of a given vertex s. This count is then divided by the maximum number of triangles that could potentially exist in the neighborhood of s, represented by $\frac{deg_n(s)(deg_n(s)-1)}{2}$. It is important to note that this formula is specific to undirected graphs. In terms of computational complexity, if an adjacency matrix is used the algorithm can be considered linear in the number of edges of the graph for each vertex s. It iterates through all the edges (v, w) of the graph and checks whether vertices v and w are neighbors of s in $O(1)$. Consequently, it would be $O(|E|)$ for each vertex s, thus $O(|V||E|)$ for the whole graph in the worst case. However, in practice the complexity is dependent on the implementation.

3.2.6 Summary

For clarity, in the rest of this thesis we will append the heuristic acronym as a suffix of the considered algorithm. For example, while McSplitLL uses the degree heuristic, McSplitLL+PR uses the PageRank heuristic. To broadly refer to a family of algorithms, regardless of the heuristic used, we will use the notation "McSplitLL+".

Table 3.1: Static heuristics for McSplit

Heuristic	**Acronym**	**Description**
Degree		Number of incident edges
PageRank	PR	Probability of being in any random walk
Betweenness Centrality	BC	Number of traversing shortest paths
Closeness Centrality	CC	Average length of shortest paths from the node
Katz Centrality*	KC*	Count of neighbors weighted by shortest distance
Local Clustering Coefficient	LCC	Cliqueness of the neighborhood

3.1 summarizes the heuristics that we will use in the rest of this thesis, with a short description of the node information used to assign the priority order. To speed up their computation, BC and CC are computed on multiple threads, since they are easily parallelizable.

Chapter 4

Parallel Architectures and Multi-Threading

Multithreading is a technique in computer programming that enables concurrent execution of multiple threads within a single process. Threads are independent sequences of instructions that can be scheduled to run simultaneously, allowing for parallel execution and efficient utilization of system resources. Each thread operates independently and can perform its own set of tasks, sharing the same memory space with other threads within the process.

Multithreading offers several advantages, including improved responsiveness and increased throughput in applications that can benefit from parallel execution. It allows for the efficient execution of tasks that can be divided into smaller, independent units, enabling better utilization of multicore processors and facilitating concurrent processing of multiple tasks.

This chapter focuses on exploring parallelization techniques for the McSplit algorithm. McSplit uses a recursive approach to perform a complete exploration of the search space, and parallelizing it poses unique challenges. The recursive nature of McSplit implies that the execution of each action is dependent on the actions that preceded it, creating interdependencies that need to be carefully preserved and managed during parallel operations. Additionally, the pruning mechanism adds another layer of complexity, as it requires maintaining the correct pruning state across multiple threads to ensure the validity of the exploration.

In this chapter, we will investigate two approaches to parallelize McSplit and improve its performance by leveraging the multithreading capabilities of modern general-purpose computers. We call these approaches McSplit Multi Branch and McSplit Branch Sharing. If these approaches reveal themselves to be effective, future research could explore the possibility of implementing a distributed version of McSplit, which could greatly increase the scalability of the algorithm and allow

it to benefit from the computational power of multiple machines.

4.1 McSplit Multi Branch (McSplitMB)

McSplit Multi Branch (McSplitMB) is a very simple parallel version of McSplit, which was built on the original idea by the authors of the original McSplit paper. In McSplitMB, the branching structure of the McSplit algorithm, as depicted in Figure 4.1, serves as the basis for parallelization. Each node in the figure represents a call to the McSplit function after selecting a node pair $\langle v, w \rangle$.

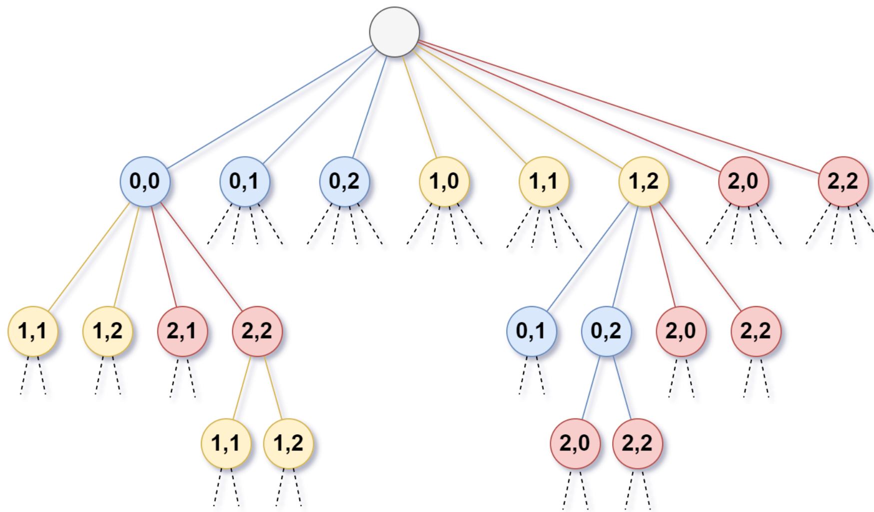


Figure 4.1: Branching structure of McSplit. In each step, the algorithm selects a node pair $\langle v, w \rangle$ and proceeds to the next step, building a tree structure.

The key idea behind McSplitMB is to split the recursive tree vertically, allowing multiple threads to concurrently explore different branches of the search space. The process starts with a master thread that performs the initial steps of the McSplit algorithm until a certain *depth*. At this depth, the master thread identifies all recursion nodes and submits them to a thread pool.

Within the thread pool, each thread is assigned to a task, which corresponds to a specific recursion node and the relative branch of the search space, and proceeds to execute the McSplit algorithm from that point onwards. Once a task is completed, the thread fetches another task from the queue. This allows for parallel computation of multiple McSplit instances, each starting from a different recursion node. By distributing the workload among multiple threads, McSplitMB significantly reduces the overall execution time, accelerating the search for the maximum common subgraph.

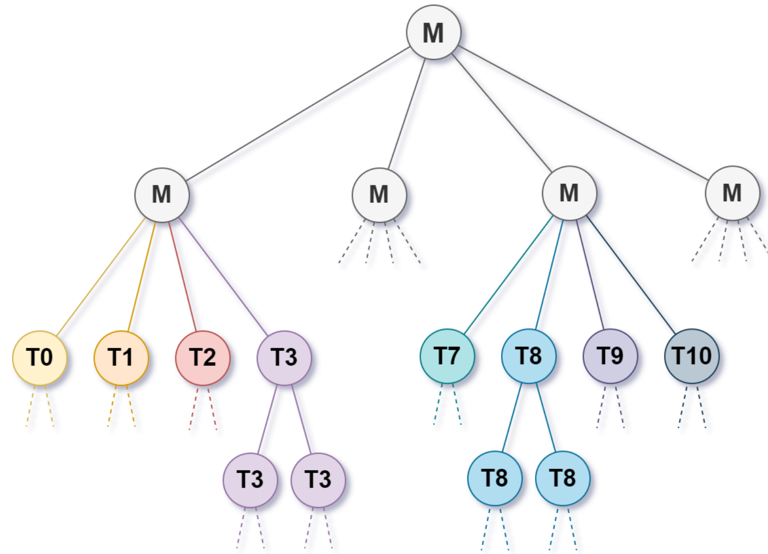


Figure 4.2: McSplitMB: the master thread M performs the initial steps of the McSplit algorithm until a certain *depth* (here $depth = 2$). At this depth, the master thread identifies all recursion nodes and submits them to a thread pool.

The number of threads involved in the parallelization is determined by the parameter $n_{threads}$, which specifies the desired level of parallelism. The *depth* parameter is used to control the granularity of the parallelization. A lower value of *depth* results in a smaller number of larger tasks submitted to the thread pool, which could reduce the thread pool management overhead. However, a lower value of *depth* also increases the risk of load imbalance among threads, as some branches of the search tree may be pruned early on, possibly leaving the task queue empty and some threads idle.

4.2 McSplit Branch Sharing (McSplitBS)

McSplit Branch Sharing (McSplitBS) is a more complex parallel version of the McSplit algorithm that addresses the potential issues of load balancing and inefficient resource utilization in McSplitMB. While McSplitMB provided a simple approach to parallelization, it could encounter challenges when dealing with workload imbalances and suboptimal branch selection. Specifically, McSplitMB could end up wasting computational resources on branches that are unlikely to lead to a maximal common subgraph. This is especially true if we employ a good heuristic for branch selection. In this case, the last threads in the thread pool could quickly finish all the tasks in the queue due to pruning, and be left idle for the rest of the execution

because incapable of picking up new branches. This would result in a suboptimal utilization of the available resources, and ultimately in a slower execution time. The issue is further aggravated by the fact that the *depth* parameter is set statically, but its optimal value is non-trivially dependent on the complexity of the input graphs. This complexity is closely related to the size and density of the graphs, but the exact relationship is not known. As a result, McSplitMB is less versatile on a heterogeneous set of inputs.

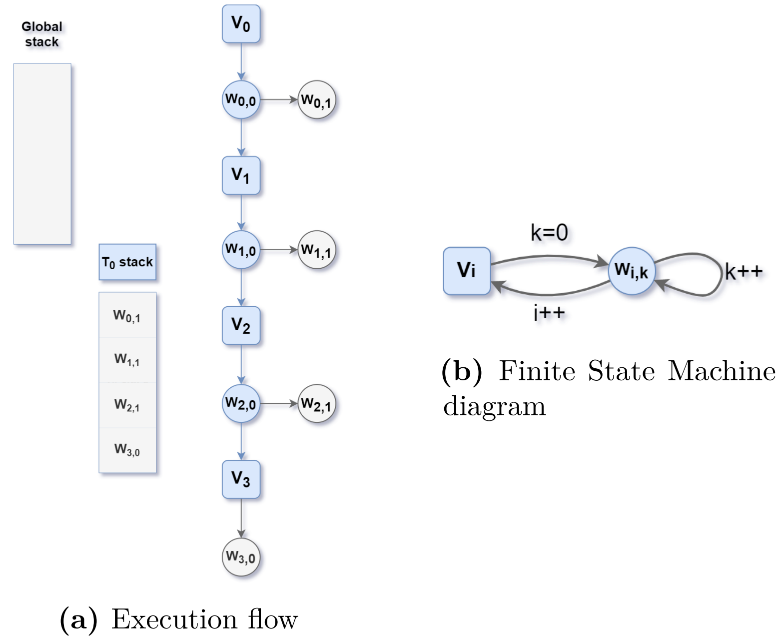
4.2.1 Building an iterative version of McSplit

To overcome the aforementioned limitations, McSplitBS introduces a transformative change to the McSplit algorithm by converting it from a recursive implementation to an iterative one. This allows for greater control and flexibility. In this new approach, the algorithm maintains a context variable called *Step*. This context variable stores all the information required for the search operations, and it allows for the transfer of search branches between different threads. By moving the *Step* object from one thread to another, we are enforcing a form of high-level context switching. The main information carried by the context is the set of bidomains $\mathbb{B}$, which implicitly defines the set of vertices that are still available for matching, plus other auxiliary information such as the current solution $S_{current}$ or the last selected vertex v. A stack data structure is used to store the context objects while they are waiting to be processed, following a Last-In-First-Out (LIFO) policy.

As a broad generalization, in the iterative implementation of McSplit in a single-thread environment the algorithm repeats the same set of operations:

1. Pop the top element of the stack.
2. Process it.
3. Push the newly created branches back to the stack.
4. Repeat until the local stack is empty.

However, due to the asymmetric nature of the `mcs()` function, which involves selecting a single vertex v and then iterating over multiple vertices w to find matches for v, the *Step* context can be of two types, $StepV$ and $StepW$, each representing part of a recursion call. $StepV$ instructs the thread to select a new vertex, and then to create a corresponding $StepW$. $StepW$ is used to select a single new vertex w, then generate a new $StepW$ object along with a new $StepV$. The creation of these objects follows a simple Finite State Machine (FSM), represented in Figure 4.3b. For ease of management, since $stepV$ objects cannot create multiple branches, they are computed immediately upon creation, and in practice, only $StepW$ elements can be pushed to the stack.



(a) Execution flow

(b) Finite State Machine diagram

Figure 4.3: McSplitBS: iterative implementation of the recursive search

Consider Figure 4.3a. Thread T_0 starts the algorithm with a *StepV* object containing the initial state of the algorithm, denoted as V_0. The object contains all the context required for the selection of a new vertex v. Upon selecting v, a new *StepW* object, $W_{0,0}$ in the diagram, is created. The *StepW* object carries additional information required for the selection of a vertex w. Once T_0 utilizes $W_{0,0}$ to select the first vertex w, it generates two new *StepV* and *StepW* objects, namely V_1 and $W_{0,1}$. $W_{0,1}$ is inserted onto the local stack, then V_1 is immediately computed, and the new $W_{1,0}$ it created is inserted as the last element in the local stack. Consequently, $W_{1,0}$ is immediately selected, while $W_{0,1}$ remains in the stack until all the child branches of $W_{1,0}$ have been exhaustively explored. $W_{1,0}$ then leads to the creation of $W_{1,1}$ (left idle in the stack) and V_2. This process continues until T_0 reaches the end of the main branch, where it is not possible to select any other vertex v. Consequently, the last *StepV* aborts its execution, and T_0 pops the last *StepW* context that was inserted and left idle in the stack, thus switching the current branch. The algorithm terminates when the local stack becomes empty, indicating the completion of the exploration of the entire search space.

4.2.2 Branch Sharing

To parallelize the iterative implementation of McSplit, a global stack is introduced, which is shared among all the threads. This global stack serves as a storage for the

StepW objects that are awaiting assignment to a thread. The operation of each thread is slightly modified as follows:

1. Pop the top element of the global stack and push it to the local stack.
   (a) Pop the top element of the local stack.
   (b) Process it.
   (c) Push the newly created elements back to the local stack.
   (d) Repeat until termination condition X.
2. Push the remaining elements from the local stack to the global stack.
3. Repeat until the global stack is empty.

The termination condition X is the policy that allows us to manage the branching behavior of the algorithm. We define it as X = *local stack is empty or local stack size is greater or equal than block_size*. The first condition is trivial: if the stack is empty, the thread has explored the entire local branch, and it can now fetch a new branch from the global stack.

The second condition is used to limit the size of the local stack, preventing the thread from exploring alone a branch that is too large. In McSplit a big branch is likely a promising one, because it has not been pruned yet. However, it is also a branch that will take a long time to explore, and it is likely that the thread will not be able to finish in time before the timeout.

Therefore, we force the thread to relinquish control of the branch and push all its pending elements to the global stack. This will effectively split the branch across multiple threads, and it will allow the algorithm to explore the branch in parallel.

Consider *block_size* = 4. The thread starts the exploration and visits the recursion tree with a DFS approach until the local stack contains $\{W_{0,1}, W_{1,1}, W_{2,1}, W_{3,0}\}$ (4.4). Then, T_0 pushes all elements to the global stack, and it immediately pops the top context object $W_{3,0}$ from it into the local stack (4.4).

As shown in 4.5, the algorithm progresses with more iterations. Let's assume the thread T_0 has popped $W_{3,0}$ from its local stack and is currently processing it, temporarily leaving the stack empty. Meanwhile, thread T_1 pops $W_{2,1}$ from the global stack and explores its local branch. If the branch is large enough, Thread T_1 will further split it, resulting in the creation of a new $W_{2,2}$ object that will be pushed back to the global stack. In this example, we assume that $W_{2,2}$ selects the last w that can be matched to the vertex v selected by V_2, so the *StepW* object only generates V_5. Since the entire branch of $W_{1,0}$ is already assigned to T_0, T_1 and T_2, thread T_3 fetches $W_{1,1}$ from the global stack.

It is notable that in Figure 4.5 $W_{0,1}$ is the only element remaining in the global stack, despite being the first element initially pushed onto it. While McSPlitMB

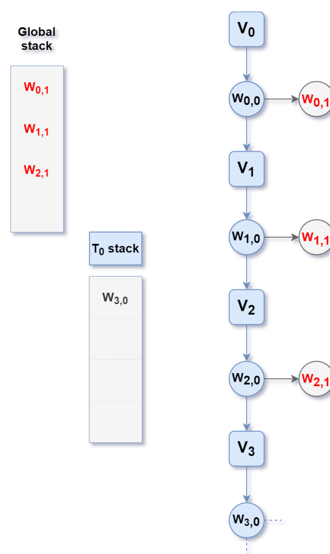


Figure 4.4: McSplitBS: global stack after T_0 reaches the termination condition

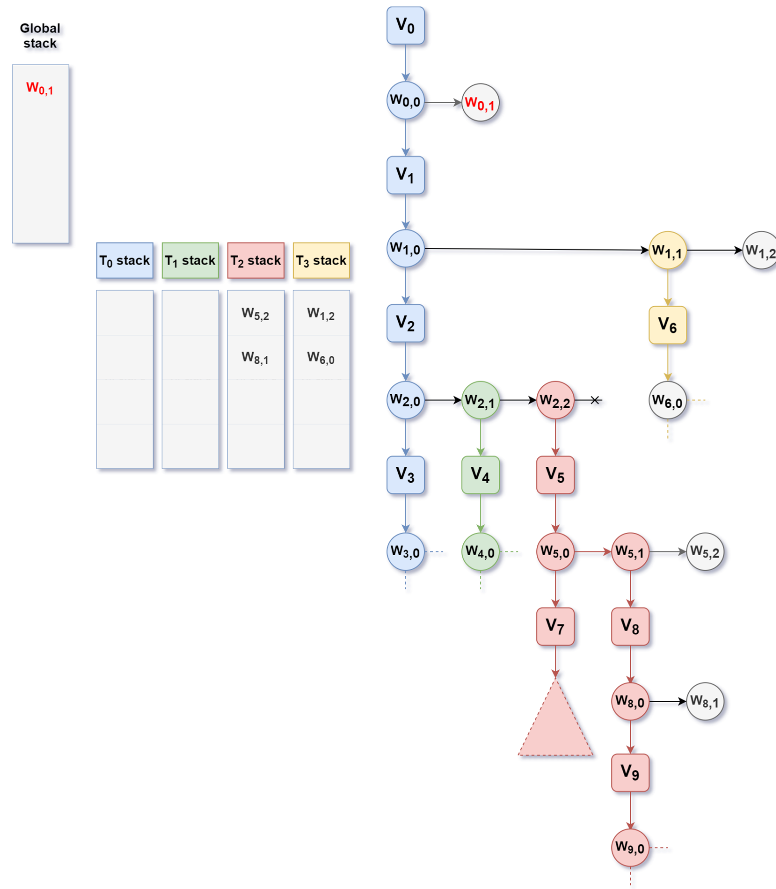


Figure 4.5: McSplitBS: subdivision in threads

performs a BFS subdivision of work, McSplitBS prioritizes the most promising branches, in order to get the largest solution as soon as possible. This highlights that McSplitBS relies heavily on the effectiveness of the node selection heuristic, as a poor branch selection at the beginning of the algorithm can significantly impact its performance.

Algorithm 11 reports a more detailed overview of the McSplitBS inner workings, considering that `mcsThread` is the function running in all $n_{threads}$ threads. For practical reasons, the actual code is slightly different from the flow described here, and the pseudocode reflects some of these changes.

Block size

The choice of *block_size* in McSplitBS has a significant impact on the granularity of the context switching among threads. A low *block_size* results in threads relinquishing their steps more frequently, leading to a larger number of steps in

```
1  S ← {}
2  Steps_global ← stack()
3  Function mcsThread(G, H)
4  |  Steps_local ← stack()
5  |  while time < timeout do
6  |  |  Pop Step sg from Steps_global under mutual exclusion
7  |  |  Steps_local.push(sg)
8  |  |  while 0 < Steps_local.size() < block_size do
9  |  |  |  s ← Steps_local.getHead()
10 |  |  |  restore context from s
11 |  |  |  if s is a StepV then
   |  |  |  |  // this is a StepV
12 |  |  |  |  if |S_current| > |S| then
13 |  |  |  |  |  S ← S_current
14 |  |  |  |  end
15 |  |  |  |  Bound ← ComputeBound(G, H, |S_current|)
16 |  |  |  |  if Bound < |S| then
17 |  |  |  |  |  return
18 |  |  |  |  end
19 |  |  |  |  β ← SelectBidomain(G, H)
20 |  |  |  |  v ← SelectVertexV(G, β)
21 |  |  |  |  G' ← G' \ {v}
22 |  |  |  |  make s a StepW
23 |  |  |  |  Steps_local.push(s)
24 |  |  |  else
   |  |  |  |  // this is a StepW
25 |  |  |  |  w ← SelectVertexW(H, β)
26 |  |  |  |  S'_current ← S_current ∪ {v : w}
27 |  |  |  |  H' ← H \ {w}
28 |  |  |  |  UpdateBidomains(G', H', v, w)
29 |  |  |  |  s_next = newStepV(context)
30 |  |  |  |  Steps_local.push(s_next)
31 |  |  |  |  if β_right = ∅ then
   |  |  |  |  |  // No more branches to explore, backtrack
32 |  |  |  |  |  Steps_local.pop()
33 |  |  |  |  end
34 |  |  |  end
35 |  |  end
36 |  |  if Steps_local.size() > 0 then
37 |  |  |  Steps_global.push(Steps_local.all())
38 |  |  end
39 |  end
```

Algorithm 11: The McSplit Branch Sharing algorithm.

the global stack. This means that the first thread quickly pushes its steps to the global stack during the initial stages of execution. The following threads will be able to pop elements from the global stack sooner, starting their exploration closer to the root of the tree. However, this increases the number of elements present on the global stack and raises the potential for concurrency conflicts among threads when fetching new context objects.

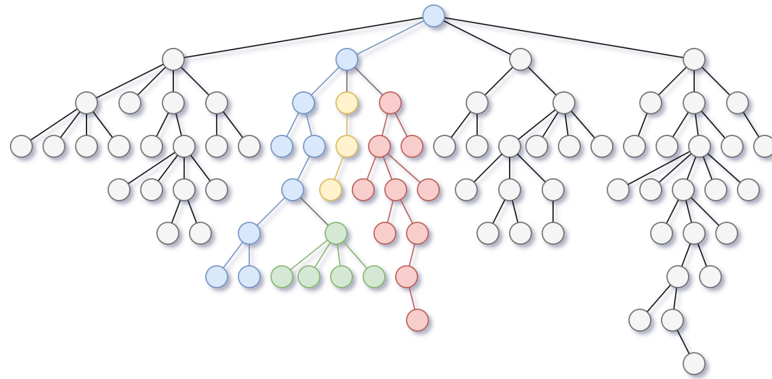
On the other hand, a high *block_size* reduces the frequency of conflicts in accessing the global stack under mutual exclusion. Furthermore, the sibling threads start their execution from deeper branches because the first thread takes longer to submit tasks to the global stack. This could be beneficial as it makes the algorithm focus on the most promising branch, which was selected by the first thread. However, if the first selected branch is not the best one, the algorithm will take longer to explore other areas of the search space. Moreover, a higher *block_size* means that the threads will hold onto their steps for a longer duration before relinquishing control, potentially leading to increased load imbalance among the threads.

Lastly, with a high *block_size* there is the possibility that a thread might dump a significant number of steps onto the global stack, potentially overshadowing all the previous steps. As a result, these earlier steps may take a considerable amount of time to be fetched and processed. This behavior shifts the algorithm's focus towards the latest branch being explored, as the steps from previous branches may experience delays in execution.

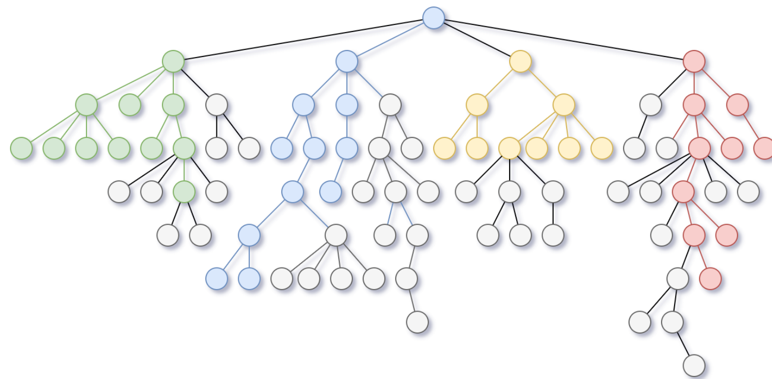
Delayed Sharing

To investigate the impact of the starting point of sibling threads on the performance of McSplitBS, we design a specialized test. In this test, regardless of the chosen *block_size*, we artificially modify the behavior of the first thread. Instead of immediately pushing elements to the global stack when the termination X is reached, we enforce a condition that prevents the first thread from pushing any elements until it performs the first pruning operation. This modification ensures that the first thread explores deeper into the first branch before sharing its progress with the sibling threads.

By delaying the initial push to the global stack, we aim to observe the effects of a more focused exploration strategy. McSplitBS typically discovers a reasonably sized solution in its first branch, especially when coupled with a good heuristic. Therefore, the pruning process primarily occurs deep within the initial branch. By restricting the first thread's ability to push to the global stack before pruning, we can determine if a tighter exploration, with a higher concentration of promising branches, leads to improved overall performance compared to the broader exploration of a low *block_size* or McSplitMB (4.6).



(a) Focused exploration



(b) Broad exploration

Figure 4.6: Using a high *block_size* or the Delayed Sharing technique results in a focused exploration, delving deep into a small set of branches, potentially missing the best branch. A broad exploration visits more branches but in a shallow manner, risking incomplete exploration of individual branches. Evaluation and experimentation are needed to determine the optimal strategy.

4.3 Conclusion

Given the complexities and trade-offs associated with the choice of McSplitMB or McSplitBS and its *block_size* parameter, it is challenging to make accurate predictions on which is the optimal strategy. The behavior of the algorithm, in terms of load distribution and branch exploration, depends on various factors such as the characteristics of the input graph, the nature of the search space, and the architecture of the underlying hardware. To determine the most suitable approach, empirical testing and experimentation are necessary. Furthermore, McSplitBS adopts a more complex implementation compared to McSplitMB, primarily due

to the new iterative structure and the introduction of the context variable *Step*, as well as the management of the frequent concurrency conflicts on the global stack. This additional complexity comes with some overhead, making it crucial for McSplitBS to deliver substantial performance improvements to justify its use. However, our primary objective is to investigate the performance trade-offs between broad and tight exploration strategies on a parallel architecture. We also aim to measure the performance boost achieved by parallelization compared to the single-threaded McSplitX+ algorithms.

Chapter 5

Graph Neural Networks (GNN)

One key challenge in McSplit is the selection of optimal nodes at each branching stage. This decision significantly impacts the efficiency and effectiveness of the algorithm. In Chapter 3 we focused on static heuristics, which assign pre-defined scores or priorities to nodes based on certain characteristics. However, applications in different domains might benefit from diverse and more specialized heuristics, which would require extensive manual trial-and-error research to identify.

To address this limitation, we turn to machine learning, aiming to leverage its capabilities to learn effective node selection policies. In particular, we explore the application of GNNs as heuristic models for the McSplit algorithm.

Our investigation begins with an exploration of an existing solution called GLSearch, which utilizes GNNs to score and select entire vertex pairs in McSplit. We will use the knowledge gained by GLSearch to develop our own in-house GNN-based models to investigate the viability and effectiveness of the ML approach as a tool to improve the node selection process in McSplit.

In the realm of machine learning, conducting research and experimentation often demands a significant investment of time and resources. Given the scope of our current endeavor, which involves exploring various potential enhancements for the McSplit algorithm, it is important to clarify our primary objective. While we aim to make advancements and identify new state-of-the-art solvers, our main focus lies in assessing the viability of using Graph Neural Networks (GNNs) as a potential solution. Through this research, we strive to contribute to the collective understanding of the utility and applicability of GNNs, paving the way for future advancements in this domain.

5.1 GLSearch

GLSearch [38] is a complex architecture that combines the power of GNNs and RL through a Deep Q-Network (DQN) architecture. While built following the McSplit framework, GLSearch applies radical changes to the node selection process. Instead of relying on static heuristics to individually select the vertices v and w, GLSearch utilizes a ML model to directly select a pair of vertices $\langle v, w \rangle$ to be matched at each branching stage. Unlike other GNN-based MCS approaches, GLSearch is still designed to explore the entire search space, given enough computational resources and time, so it is still classified as a ground-truth solver for the MCS problem.

5.1.1 The architecture

During the execution of GLSearch, at each branching point in the McSplit algorithm we encounter a specific state representing the current bidomains and a set of multiple actions representing the possible pairs of vertices that can be selected. This state and actions could be stored in a Q-table in the same way as McSplitRL, and use Reinforcement Learning to predict the best match for a given state.

However, in McSplitRL the selection of the vertices was essentially stateless, and in McSplitLL the selection of w had as a state the single vertex v, so the Q-table had a memory occupation complexity of $O(|V|^2)$. When dealing with medium to large graphs, the number of possible combinations of bidomain states and possible vertex matches becomes exceedingly large, making it impractical to store all of them in a Q-table. To overcome this issue, GLSearch adopts a Deep Q-Network (DQN). A DQN is a reinforcement learning technique where the Q-table is approximated by a MLP model. Instead of explicitly storing the Q-values, the DQN learns to approximate them through training and saves a model object of a much smaller size.

As seen in Section 2.5, MLPs require a fixed-size input in the form of a vector, while bidomains take the form of collections of nodes of variable size. This requires the definition of a mapping function capable of generating a state-action pair representation that can be fed to the DQN. To achieve this encoding, GLSearch utilizes a Graph Attention Network (GAT). The GAT is run on the set of bidomains $\mathbb{B}$ at the current state of the McSplit recursion to compute a fixed-size embedding $e_{\mathbb{B}}$. To better represent the state, the GAT is also used on the entire graphs G and H to compute the global embeddings e_G and e_H. To represent the actions, the GAT computes node-level embeddings e_v and e_w for v and w, which are the vertices of the pair considered by each action.

These embeddings are then combined using a one-dimensional convolution operation, resulting in a final fixed-size vector embedding that encodes both the state and the considered action. The resulting representation is fed into the MLP

model, which outputs a goodness score $V(\mathbb{B}, \langle v, w\rangle)$ for the considered state-action pair.

Lastly, the authors introduce a discount factor γ to adjust the importance of the goodness score. This is done to increase the difference between the rewards and hopefully optimize the training of the model.

The architecture of GlSearch is quite complex, and it includes other small details that we will not cover in this thesis. For a more in-depth explanation of the model, we refer the reader to the original paper [38].

Overall, the entire DQN pipeline can be expressed using the following equation:

$$\begin{aligned} Q(B, \langle v, w\rangle) = & 1 + \gamma \cdot MLP(conv1d(\\ & GAT(\mathbb{B}), GAT(G), GAT(H), GAT(v), GAT(w))) \end{aligned} \tag{5.1}$$

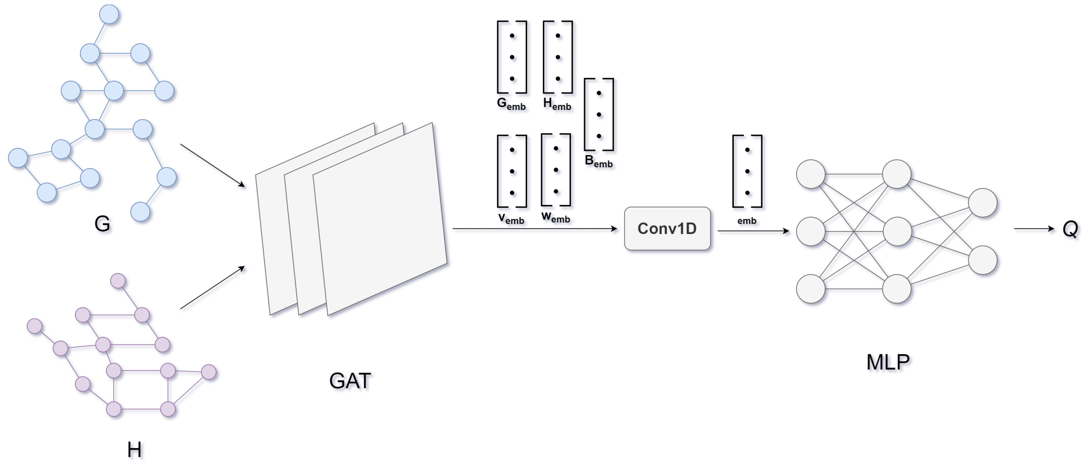


Figure 5.1: GLSearch architecture

5.1.2 Training

Training GLSearch poses several challenges, particularly in determining which state-action pairs are favorable or unfavorable. To train the DQN, it is necessary to have a function ENV() that simulates the environment response to a given state-action pair. In this case, it should return a score of how good a branch is. The crux of the issue lies in the fact that the MCS problem typically requires significant time and computational resources to obtain the optimal solution. Consequently, it is often difficult to ascertain which branches within the search space are genuinely advantageous, because we don't know in which branch the optimal solution lies.

To overcome this challenge, GLSearch incorporates a curriculum learning approach. Curriculum learning involves gradually exposing the learning model to increasingly complex or challenging instances of the problem. In the context of GLSearch, this approach helps in guiding the training process, starting from simpler instances where the optimal solutions are known and gradually progressing towards more challenging scenarios.

Furthermore, the training is split into 3 stages:

1. PRE-TRAIN: During the initial stage of training in GLSearch, the model begins with no prior knowledge. To establish a strong foundation for the subsequent learning process, training is initiated using small graphs for which it is computationally feasible to calculate the optimal solution. To obtain the optimal solution, the McSplit algorithm is employed to perform a complete traversal of the search space without employing any pruning techniques.

   By exhaustively exploring the search space, the pre-training phase allows for the determination of the exact length of each branch and the identification of the best solution within that branch. As a result, during the training process, a label is assigned to each state-action pair based on the remaining length y_t of the best solution within the corresponding branch. Then the DQN loss function is replaced with a MSE loss function L to train the predicted score $Q(\mathbb{B}, \langle v, w \rangle)$ to match the label y_t:

$$L = \frac{1}{N} \sum_{i=1}^{N} (Q(\mathbb{B}, \langle v, w \rangle) - y_t)^2 \tag{5.2}$$

2. IMITATION LEARNING: In the second stage of training, GLSearch goes beyond using only small graphs for training and incorporates larger graphs as well. The reason for this is to ensure better generalization of the model on larger graphs, as small graphs may not fully capture the complexity and challenges presented in real-world networks.

   Since the optimal solution is not available, GLSearch leverages McSplit as an expert solver for the MCS problem. McSplit guides the search process by making informed decisions at each branching point, based on the default degree heuristic.

   The DQN in GLSearch is trained again using a MSE loss function, where the labels are obtained from the decisions made by McSplit. In other words, McSplit's branching decisions act as the ground-truth labels for training the DQN. The objective is to train the model to replicate the decision-making process of McSplit and ultimately achieve similar performance, rather than surpassing it.

3. RANDOMIZED LEARNING: In the final stage of training, the objective is to train the model to make smarter decisions than those made by McSplit. This is done by performing the training using two parallel decision-making agents: the DQN and a random policy.

   A parameter ϵ determines the probability of using the random policy to select the next branch of the recursion. This is done to ensure that if the model is not able to make good decisions yet, a stochastic agent allows for the exploration of new and potentially promising branches that the DQN might have overlooked. Assuming that the model improves in the course of this stage, ϵ is set to gradually decay, so that by the end of the training the DQN is the primary decision maker in the algorithm.

5.1.3 Our experience

GLSearch is a complex system, and due to the complexities involved, we have omitted several details in this overview. Our primary objective was to understand and potentially adapt GLSearch methodologies for our experiments, as our main goal was to explore the application of GNNs to McSplit. Consequently, we dedicated a significant amount of time to comprehend the code implementation. However, it is important to note that the publicly provided code had several flaws that are still unresolved at the time of writing. Part of our effort on GLSearch has been dedicated to the reformatting and restructuring of the code, in the hope to obtain a fully working system. Regrettably, this work was for the most part unsuccessful, and completely rewriting from the start such a complex body of code was beyond the scope of our research, so we were limited to utilizing what had been provided.

We were able to successfully execute the program on a very limited set of graph pairs and evaluate the available pre-trained models, but we encountered challenges in getting the training code to function properly. As a result, we are unable to conduct comprehensive testing of the software. Most importantly, GLSearch has been trained to tackle the MCCS problem, whereas our focus is on the MCS, so it is not possible to perform a fair comparison of GLSearch against our other models. Considering these difficulties, we made the decision to discontinue our exploration of GLSearch and move on to building our own models.

5.2 McSplitGNN

The first of these models is called McSplitGNN. Our objective is to develop a simpler system compared to GLSearch, which will enable us to gain a deeper understanding of how GNNs behave in the context of the MCS problem. However, unlike GLSearch, we aim to preserve the fundamental structure of McSplit, where the selection of

vertices v and w is performed separately. Nonetheless, the architecture can be extended to the selection of a matching pair $\langle v, w \rangle$, if needed.

The simplest application of the GNN is the individual selection of a vertex v or w. In the traditional McSplitX+ models, this is done through a combination of RL and static heuristics. In McSplitGNN, we replace the static heuristics with a dynamic GNN-based model, which is trained to assign a score to each candidate vertex.

5.2.1 The architecture

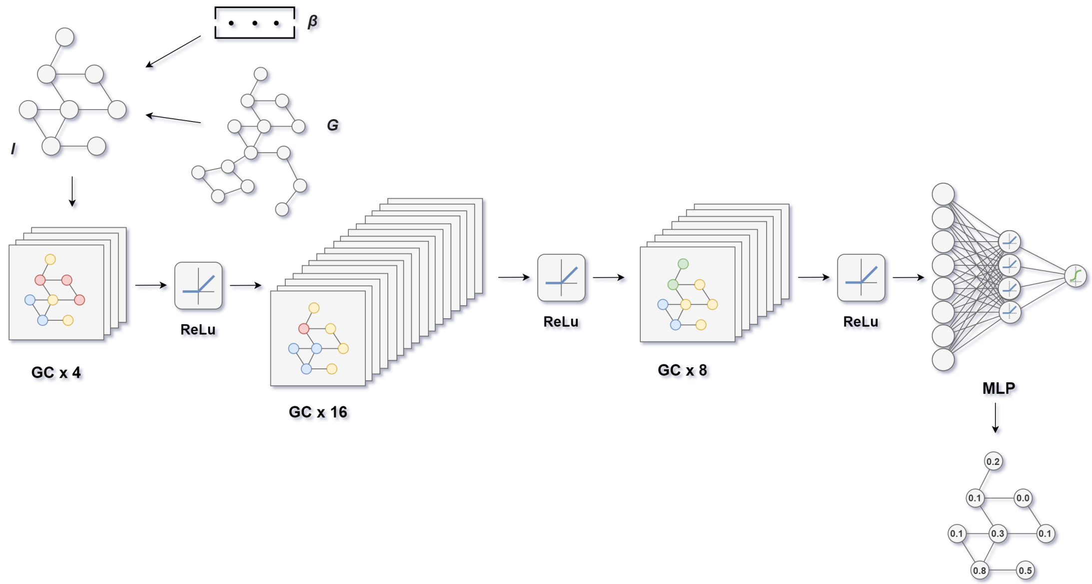


Figure 5.2: McSplitGNN architecture

The architecture of this model is shown in Figure 5.2. The input is the selected bidomain β that contains the candidate vertices u. Since we are working on individual vertex selection, in this section we will use only one side of the bidomain, but the process is identical for both sides. The bidomain, which is internally represented as a collection of vertices with the same bidomain label, must be converted into an induced subgraph $I \subseteq G$ before being fed into the GNN. The GNN is a GCN composed of 3 Graph Convolutional layers. Given a starting node-level embedding of size 1, the successive layers produce embeddings of sizes 4, 16 and 8. The dimension of the embeddings directly impacts the number of model parameters and, consequently, the training time. Considering that bidomains typically represent small subgraphs with limited connectivity, thus having low complexity, we opted for a smaller number of parameters in favor of being able to

run more training epochs.

The final embedding is fed into a two-layer MLP which compresses each node-level embedding in a single scalar score. As in the convolutional layers, the MLP uses ReLU as the activation function in the first layer, and the final score is normalized by a sigmoid function. The output of the MLP is a vector of size $|\beta|$, where each element represents the score of the corresponding vertex in the bidomain. The node with the highest score is selected as the next vertex v or w in McSplit.

5.2.2 Training

The main challenge of integrating GNNs into McSplit is the scarcity of training data. Our model is supposed to work like a heuristic value function, and we opt for a dynamic heuristic approach such as GLSearch, such that in the testing phase the model is used at each branching point to get a localized set of scores. Our focus is to have a fast training process, in order to be able to test different architectures and hyperparameters. For this reason, we decided to use a destructured dataset, which is generated offline before the training process.

To generate such a dataset, the McSplit algorithm is run on a set of graphs, and the context of each branching point in the algorithm is saved to a binary file. The resulting context objects contain the currently selected bidomain and the vertex scores ν_i. These scores are obtained by taking advantage of the recursive nature of McSplit: during a branching event, the algorithm calls itself to explore the new branch, and when the function invocation returns, we can determine if the branch contributed to increasing the current solution $S_{current}$, and if so, by how much.

By relying on McSplit, we cannot expect the model to surpass its performance in terms of solution size. Furthermore, by working as a dynamic heuristic, the algorithm is necessarily slower than the McSplitX+ models, which use static heuristics that are computed only once at the start of the exploration. Therefore, we do not expect the model to be able to make better decisions than McSplit, but if it shows comparable performance it would mean that the model could be improved in future research using a progressive curriculum approach.

It is important to note that the McSplit is run with pruning enabled. Consequently, the pruned branches will achieve a score of $\nu_i = -1$, or at least less than their actual maximum increase. While this could be addressed by assigning the bound as the score of the pruned branches, this could unfairly advantage the pruned branches, as they would be assigned a higher score than the actual increase they would have achieved. Regardless, our strategy does not heavily interfere with the goodness of the scores ν_i, as the pruned branches in McSplit are guaranteed to not have the maximum solution, but it means that the scores are not probabilistic. For this reason, we use a sigmoid function as the final activation function of the

MLP, instead of a softmax. The objective of the model is therefore not to predict the probability of a branch being the best, but rather to try to output a holistic score that considers the branch length and whether the branch will be pruned.

The scores ν_i are then further multiplied by the factor α_i, which indicates if the vertex is in the final solution found by McSplit. This adjustment penalizes branches with a good increase in solution size, but which include matchings that ultimately prevent the algorithm from finding the maximum solution.

The resulting dataset is a set of individual branching records. After shuffling, this allows the model to learn patterns that are independent of the execution order of the branching events. The training is done using the Adam optimizer, with a learning rate of 0.005, and the MSE loss function is employed to compare the sigmoid σ of the label with the predicted reward y_i, for each vertex $u_i \in \beta$:

$$\begin{aligned} \alpha_i &= \begin{cases} 1 & \text{if } u_i \in S \\ 0 & \text{if } u_i \notin S \end{cases} \\ \mathcal{L} &= \frac{1}{n} \sum_{i=1}^{|\beta|} (\sigma(\alpha_i \nu_i) - y_i)^2 \end{aligned} \tag{5.3}$$

5.3 McSplit DiffGNN

McSplitGNN is designed as a dynamic heuristic score generator, to imitate the behavior of a static heuristic. However, this involves losing graph information during the compression transition from an embedding to a scalar value performed in the MLP stage. As such, we create a new differential model, DiffGNN, which is designed to perform decisions based on a difference of vector embeddings.

5.3.1 Model Architecture

The architecture (5.3) accepts as input the two sides of a bidomain, in the form of two graphs β_{left} and β_{right}. The bidomain graphs are processed by a GCN module similar to the one used in McSplitGNN. However, this GCN is composed of 4 convolutional layers, of respective sizes of 4, 16, 32 and 64. By increasing the number of layers, we are collecting more information from a wider neighborhood of the nodes. Additionally, increasing the size of the final layers allows the model to store more information, despite costing more computational power. The GCN uses a ReLU activation function internally, and a sigmoid activation function on the final layer.

The GCN module is run twice on β_{left} and β_{right} to produce two collections $e_l \in \mathbb{R}^{|\beta_{left}|,64}$ and $e_r \in \mathbb{R}^{|\beta_{right}|,64}$ of node-level embeddings. The objective of the

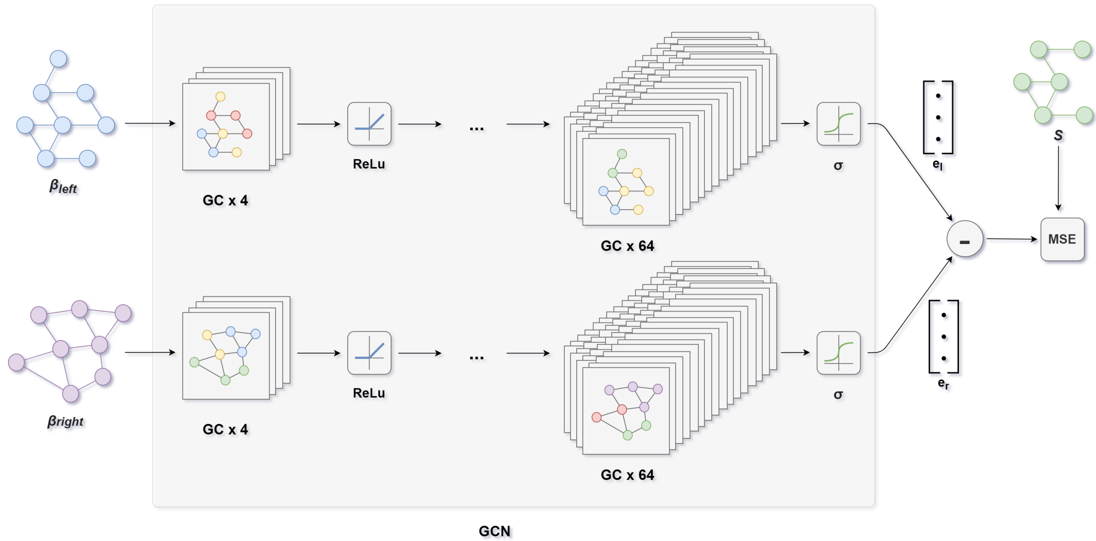


Figure 5.3: DiffGNN architecture

model is to assign similar embeddings to nodes v and w that should be matched together, and dissimilar embeddings to nodes that should not be matched together. To achieve this, the algorithm computes the differences $d_{v,w} \in \mathbb{R}^{64}$ of all the possible combinations of the e_l and e_r embeddings, resulting in 3D tensor $d \in \mathbb{R}^{|\beta_{left}|,|\beta_{right}|,64}$.

The pair inference process consists in selecting v using the DAL policy with the PageRank heuristic. Then, the algorithm selects the pair $\langle v, w\rangle$ that minimizes the MSE of the difference $d_{v,w}$, thus selecting the vertex w. Although the GNN is currently designed to select the vertex w, once the model is perfectioned and properly trained, it could be adapted to select both v and w at once as a single pair, by selecting the two closest matchings across all possible combinations. This would allow the model to be used as a standalone dynamic heuristic, without the need for the DAL policy. On the other hand, selecting a pair in a single step in a bidomain of a thousand nodes would require comparing a million pairs of vector embeddings, which is computationally expensive. As such, we leave this as a future work.

5.3.2 Training

The training process for our model closely follows that of McSplitGNN, as described in Section 5.2. After extracting the context of the branching points of a regular McSplit run, which includes the bidomain graphs β_{left} and β_{right}, the model is trained using the differences of the embeddings generated by the GCN module.

To update the trainable parameters, a MSE loss function $\mathcal{L}$ is applied to the

tensor d, asking each difference to be 0 if the corresponding nodes are matched in S, 0.5 otherwise. We ask for a difference of 0.5 because the embeddings are the output of a sigmoid function, so pushing them too far apart might lead to a vanishing gradient situation.

$$l_{v,w} = \begin{cases} [0] \in \mathbb{R}^{64} & \text{if } \langle v, w \rangle \in S \\ [0.5] \in \mathbb{R}^{64} & \text{if } \langle v, w \rangle \notin S \end{cases} \tag{5.4}$$
$$\mathcal{L} = \frac{1}{|\beta_{left}| * |\beta_{right}| * 64} \sum_{v=1}^{|\beta_{left}|} \sum_{w=1}^{|\beta_{right}|} \sum_{k=1}^{64} (d_{v,w,k} - l_{v,w,k})^2$$

During training, it can be observed that the distribution of these labels $l_{v,w}$ is highly unbalanced, with the majority of combination matchings $\langle v, w \rangle$ not appearing in the solution S. To address this imbalance, a filter is incorporated into the training pipeline. This filter randomly samples k non-matching combinations, where k is the number of matching combinations. This approach ensures a balanced dataset for training and prevents the model from learning to always output a score of 0.5. Alternatively, a weighted loss function could have been used, but the filter offers the advantage of significantly reducing the amount of data needed to be computed in Equation 5.4 by several orders of magnitude, particularly for large bidomains.

5.3.3 Training on synthetic data

One of the challenges of supervised Machine Learning is to identify and collect an appropriate set of labels for the training data. In the case of McSplitGNN, the labels are mostly based on the decision-making process of McSplit, filtered by the matchings of the solution S. In DiffGNN, the labels are immediately dependent on the solution, which is used to identify the pair combinations $\langle v, w \rangle$ that should be rewarded. However, due to the high algorithmic complexity of the MCS problem, it is not feasible to compute the optimal solution for each pair of graphs in the dataset, so the labels are generated using a suboptimal solution.

The issue becomes considerably more relevant on larger graphs, where the McSplit algorithm might find a much smaller set of matchings than the actual maximum common subgraph. As such, the model is likely to be trained on a set of labels that is not representative of the actual optimal solution and therefore of the actual best decision-making process.

To address this issue, we propose to generate a synthetic dataset of large graph pairs and their relative solution. One of the key challenges in this process is to devise an algorithm or procedure able to synthesize data that follow a similar feature distribution compared to real-world test instances. Our approach is therefore the following:

1. Fetch a Graph G from a pool of real-world graphs and create two copies G_1 and G_2.

2. Inside each copy G_i, attempt to attach a new node v to each vertex u with a probability p_v.

3. Inside each copy G_i, attempt to create every possible non-existent edge (u, v) with a probability p_e, where u and v cannot be both vertices of G.

4. Randomly remap the vertex labels of G_1 and G_2 and save the pairs of remapped labels of the node in G as the solution S.

The procedure creates two new graphs G_1 and G_2 such that the original graph G is the maximum common subgraph of the new pair. The probability p_v is used to control the number of new vertices added to G, and it controls how big the solution will be relative to the two synthetic graphs. The probability p_e is used to control the number of edges added to G, and it controls how connected the new graphs will be. To ensure a good quality of the generated data, the value of p_e should be assigned to match the connectivity of the original graph G. This step cannot create new edges between vertices of the original graph G to ensure that S will be an induced subgraph. Additional checks can be enforced to make sure that the new elements do not increase the size of the actual optimal solution (as an example, adding a single leaf node f to the same vertex of G_1 and G_2 would make f a part of the maximum common subgraph). However, internal testing showed that this suboptimal solution S is already considerably larger than the solution found by McSplit, so this step will not be taken in our experiments. The vertex remapping is done to ensure that no information about the original graph is retained in the new pair.

The procedure is repeated for each graph in the dataset, and the resulting pairs are saved as the synthetic dataset. Using randomized probabilities ensures that the generated data will follow a similar distribution to the real-world data, but it will not follow a specific pattern. This is important to avoid overfitting the model to a specific type of graph.

Chapter 6

Experimental Analysis

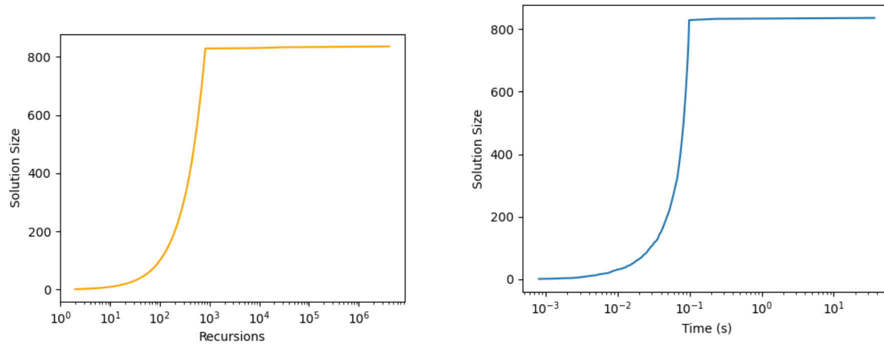
In this chapter, we test the behavior of our proposed algorithms and present an experimental analysis of our study, focusing on the performance evaluation of the different approaches applied to the MCS problem. We start by discussing the testing setup, including the dataset and hardware configuration. Next, we individually analyze the results of the static heuristics, parallel architectures, and GNN-based models. We examine their performance, discuss their strengths and limitations, and highlight any notable observations. Finally, we compare the best-performing approaches across the different families, aiming to identify the most promising solutions for improving the efficiency and effectiveness of the McSplit algorithm in solving the MCS problem.

6.1 The experimental setup

Applications of the MCS problem can be found in several domains, and our testing methodology aims to simulate such real-world scenarios. To achieve this goal, it is essential to carefully select the datasets and design the testing apparatus to reflect the research objectives.

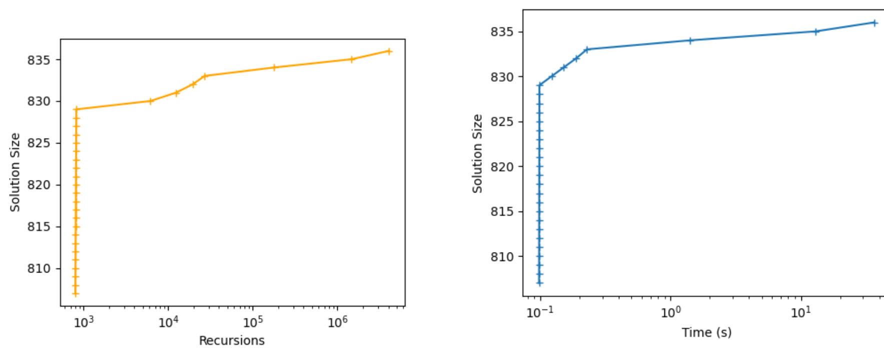
6.1.1 Testing methodology

Due to the NP nature of the problem, there are no real guarantees of finding an optimal solution in a short amount of time, so the goal of this research is to find an algorithm that searches for a best-effort solution within a limited timeframe. This property is particularly important when dealing with multiple graph pairs and the need to find multiple MCSs. In such scenarios, spending excessive time on each pair is not feasible, and an efficient algorithm that delivers good solutions in a short time becomes crucial.



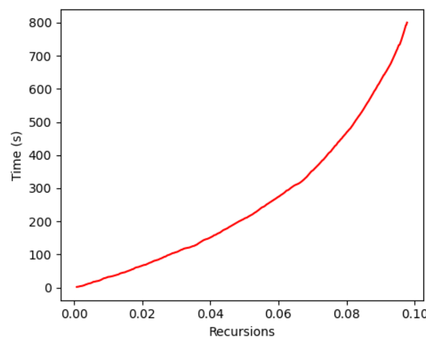
(a) Size of the largest solution during recursion

(b) Size of the largest solution over time



(c) Detail of Figure 6.1a

(d) Detail of Figure 6.1b



(e) Number of recursions over time in the linear descent phase

Figure 6.1: Progression of the best solution size ($|S|$) in McSplit. The plots are in semi-log scale. 6.1c and 6.1d show a zoomed-in version of the plots in the last stages of the algorithm.

Consequently, all tests will be run with a strict timeout, which should be long enough to allow for the visit of at least a few branches. 6.1 reports the progression

of the size of the best solution ($|S|$) produced by McSplit, for a single run of the algorithm on a pair of graphs of 1945 vertices with a timeout of 3 minutes. The plots are in semi-log scale, and show that the algorithm is able to find a good solution in less than one second after a quick descent in quasi-linear time, after which the solution size does not appreciably increase anymore. Consequently, in all tests the timeout will be set to 60 seconds, to account for the longer visit times of the largest graphs in our test suite. The timeout does not take into account the graph loading times.

Interestingly, Figure 6.1e highlights that the recursion speed is not constant. This is due to the fact that while descending the search tree, the bidomains get smaller and their management is faster.

All tests have been carried out on a workstation equipped with an Intel Core i9-10900KF CPU with 64 GB of RAM, under the Ubuntu 20.04 LTS Focal Fossa operating system. The C++ code for the McSplitX+ algorithms is written in C++ and compiled with GCC 9.3. The GNN-based models are written in Python 3.8 using PyTorch 2.0.1 and CUDA 12.1. The training process was carried out on an NVIDIA GeForce RTX 3070 GPU with 8 GB of VRAM.

6.1.2 Datasets

Datasets for the MCS problem are somewhat difficult to obtain. This is because we need pairs of graphs that are structurally similar, but not isomorphic. This is to simulate the most common use cases, where the two graphs are contextualized in the same domain (e.g., two molecules, two road networks, two social networks, etc.), and they share a similar structure.

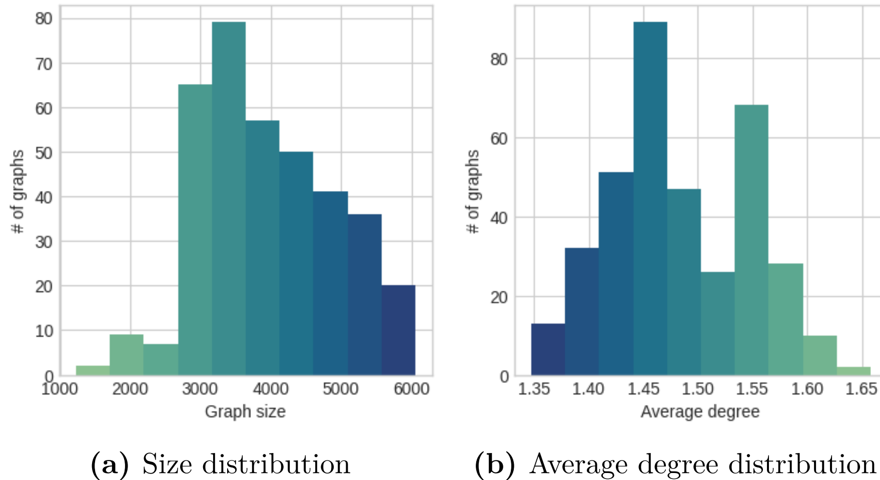
We aim to use moderately sized graphs. This is done primarily to stress test the algorithms on harder problems and to ensure that eventual differences in performance have enough time during the search to manifest. On the other, we want to avoid using graphs that are too large, as they would require excessive time to solve, which is not feasible for our extensive testing requirements.

Our test suite will be composed of three datasets:

1. Small: A collection of graph pairs from a public MCS dataset [39]. The original dataset contains 54,600 small graph pairs with different characteristics, from which we sampled 400 graph pairs, all composed of 100 nodes or fewer. This dataset is quite homogeneous in terms of size, but its difficulty varies significantly. The connectivity is between 10% to 90%. This dataset is used to test the algorithms on a large number of smaller graph pairs, to obtain a broader statistical overview of their performance.

2. Large: A collection of graph pairs derived from the AS-733 dataset, collected from Stanford University[40]. It contains 733 real-world graphs relative to the

Oregon Autonomous Systems (AS). These are networks of internet routers spanning the years 1997 to 2000. The dataset contains single graphs of size between 2,000 and 6,000 nodes. These graphs were sorted by size and then paired together two-by-two, to achieve 366 pairs with a maximum size ratio of 10%.



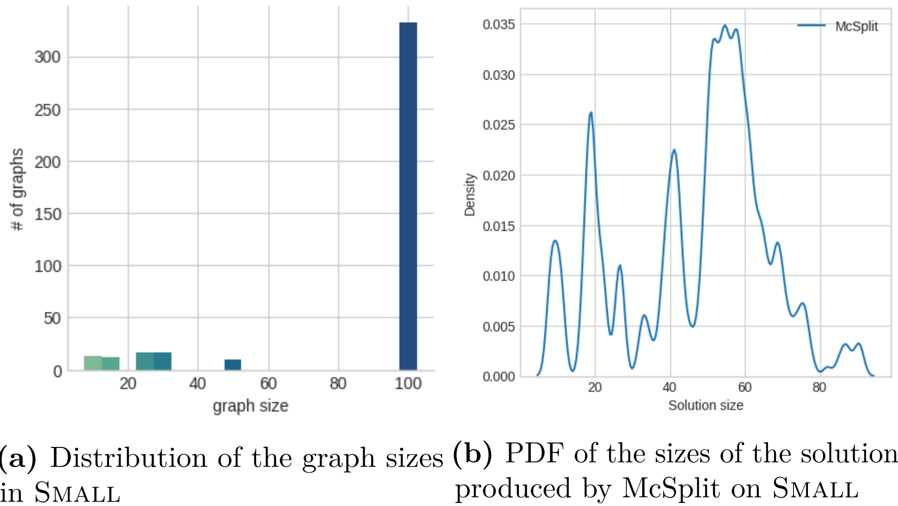
(a) Size distribution **(b)** Average degree distribution

Figure 6.2: Statistics of the LARGE dataset

The distribution of the graph size, reported in Figure 6.2a, follows a relatively wide distribution, with most graphs having a size between 3,000 and 5,000 nodes, which will have to be taken into account during the analysis. The average node degree hovers around 1.5, with less overall variation (Figure 6.2b). This dataset is used to test the algorithms on a set of large heterogeneous real-world graph pairs, to evaluate their performance on more challenging and varied problems. The increased size also helps to better evaluate the performance of the parallel algorithms, which explore a larger search space.

3. LARGE-FINETUNING: The previous datasets contain 400 and 366 MCS instances each. Assuming a 60 seconds runtime for each graph pair (which is a conservative estimate that is not considering the graph loading times and other test management overhead), each complete run of the algorithm requires more than six hours. Considering the large amount of testing mandated by the high number of algorithms and hyperparameters, this is not a feasible approach. To overcome this issue, LARGE-FINETUNING is a sampled subset of graph pairs of the LARGE dataset, containing 122 pairs. The sampling was done through a uniform distribution over the size-sorted pairs, to ensure that the distribution of the graph sizes is preserved. This dataset cuts the testing time to about 2 hours per experiment.

6.1.3 Result post-processing



(a) Distribution of the graph sizes in SMALL

(b) PDF of the sizes of the solution produced by McSplit on SMALL

Figure 6.3: McSplit performance on SMALL dataset

We run McSplit on SMALL to analyze its behavior. The size of the best solutions found by the algorithm is highly variable, depending on the graph size and the complexity of the problem. Figure 6.3b reports the Probability Density Function (PDF) of the sizes of the solution produced by McSplit on SMALL. For reference, Figure 6.3a shows the distribution of the graph sizes on the same dataset. A large majority of test instances are of size 100, while a few outliers are inserted to get an idea of the behavior of the algorithm on very small graphs and detect possible anomalies. The PDF of the solution sizes has a few well-defined peaks for the smaller graphs, approximately around 10 and 20 nodes, but it is much more variable for the larger graphs, where McSplit cannot explore the entire search space. Specifically, the graphs of size 100 have solutions that range from 50 to 90 nodes, with a peak around 55 nodes.

This is a problem, as it highlights that the complexity of the problem varies significantly across the instances, and that the average graph size is not necessarily a good indicator of the difficulty of the problem. Furthermore, the observed results support our decision to use datasets with a large number of instances, to obtain a more accurate statistical estimate of the performance of the algorithms that is independent of such drastic variations.

Consider two graph pairs A and B of the same size, but where A is more difficult than B. When evaluating multiple algorithms on these instances, both will produce smaller solutions for A and larger solutions for B. Therefore, an absolute difference of k nodes between the solutions of the two algorithms on A is much

more significant than the same difference on B. Consequently, to analyze the data we will not be able to rely on a simple average of the sizes of the solutions, as it would fail to capture the complexity of the problem. Instead, we will use a set of post-processing techniques to obtain a more accurate representation of the results.

Gain plots To compare the performance of multiple algorithms, several post-processing techniques will be applied to the obtained set of solution sizes:

- **Sorting**: The results are sorted by increased solution size. If multiple algorithms are being compared, the sorting can be performed relative to just one of these algorithms, or relative to the average solution. This behaves as a proxy metric for the complexity of the problem, as the instances at the left of the graphs represent harder problems with overall smaller solutions.

- **Normalization**: The size of the solution has considerable variations depending on the graph size and complexity. To achieve a better visualization, for each instance the results can be normalized to the result of a single algorithm, or to the average solution size. This allows us to compare the relative performance of the algorithms on each instance. This technique is particularly useful on the Large and Large-Finetuning datasets, where the graph size varies significantly.

- **Rolling Average**: When comparing multiple algorithms, the normalized data can still highly variable, since an instance can be easy for one algorithm and hard for another. To obtain a smoother visualization, the results can be averaged over a window of instances. Assuming a set of N MCS solutions S and window size of $W < N$, we obtain a collection of $N - W$ values r_i, where r_i represents the average over a contiguous set of M results.

$$r_i = \frac{1}{W} \sum_{j=i}^{i+W} |S_j| \qquad \forall i \in [1, N - W] \tag{6.1}$$

  While this technique hides the possible outliers and reduces the visible variance, it allows us to better visualize the overall trend of the algorithms.

The transformed data will therefore be charted on a line plot to show the relative performance of the algorithms. The x-axis will represent the windows of sorted instances produced by the rolling average, while the y-axis will represent the average normalized solution size of each window.

Mean Normalized Difference (MND) While the line plots are useful to visualize the relative performance of the algorithms, they do not constitute a single

numerical metric to quantify the performance difference between the algorithms. Consequently, we devise the following function, which will be called Mean Normalized Difference (MND). MND is defined as the average of the difference between the solution sizes of two algorithms, normalized by their average. Formally, given two distributions of N value, A and B, the MND is defined as:

$$\begin{aligned} \mathrm{MND}(A, B) &= \frac{100}{N} \sum_{i=1}^{N} \frac{A_i - B_i}{\frac{A_i + B_i}{2}} \\ &= \frac{100}{N} \sum_{i=1}^{N} 2 \frac{A_i - B_i}{A_i + B_i} \end{aligned} \tag{6.2}$$

In the formula, the result is multiplied by 100 to obtain a percentage value. This metric is used because the solutions sizes are highly variable, so a regular average of differences would be skewed towards the instances with a large solution size. The MND intuitively represents a distance metric between two algorithms A and B, following similar but unknown distributions. Furthermore, it is signed and symmetric, so it also carries the information on which algorithm is more performant than the other. These scores will be computed for each pair of algorithms and shown in a heatmap, to complement the insights offered by the line plots.

6.2 Experimental Analysis of the Static Heuristics

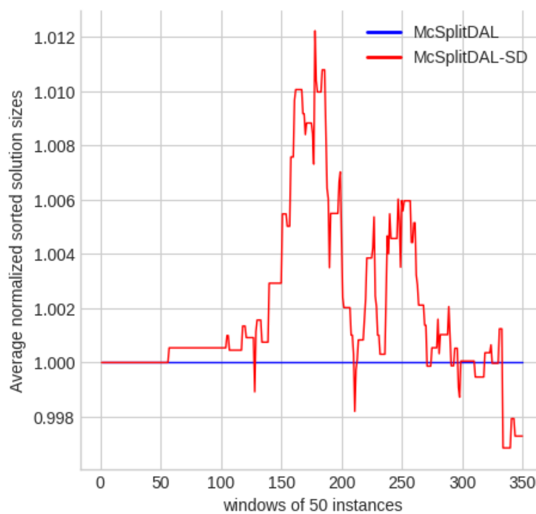
Designed the testing framework, we can now proceed to the experimental analysis of the algorithmic optimizations described in Chapter 3.

6.2.1 McSplitDAL implementations

In this section, we will compare the performance of the different implementations of McSplitDAL, introduced in Section 3.1.2. We will use the SMALL dataset due to its large instance number, hoping to detect the small statistical differences between the different implementations.

McSplitSD

We first validate the performance increase of the SD policy proposed by Trimble [23].



(a) Rolling average ($W = 50$), normalized by McSplitDAL, sorted by average.

Method	MND (%)	Average
McSplitDAL	0.00%	45.605
McSplitDAL-SD	**0.11%**	45.650

(b) Test statistics. The MND is relative to McSplitDAL.

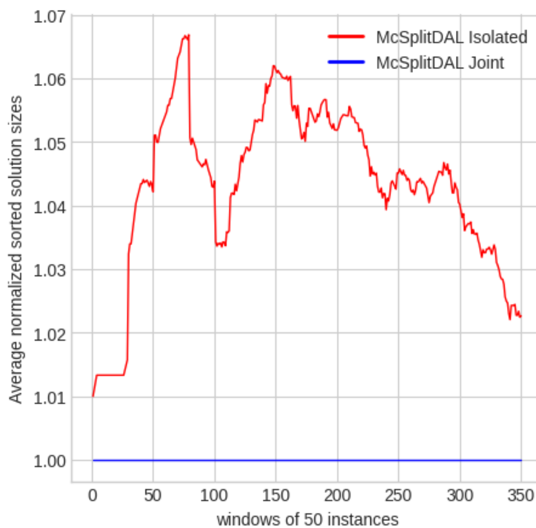
Figure 6.4: Comparison of McSplitDAL and McSplitDAL-SD on SMALL

The gain plot in Figure 6.4a shows the rolling average of size 50 of the solution sizes normalized to the results of McSplitDAL. Therefore, the plot of McSplitDAL is a constant line at 1.0, while the line of McSplitDAL-SD shows the relative performance of the SD policy. Predictably, the smaller instances to the left of the

graph have all the same solution size, since the algorithm is able to explore most, if not all, the search space.

According to the results, the SD policy does improve the performance of McSplit-DAL, but not by a considerable margin. Over the SMALL dataset the McSplitDAL-SD achieves just a 0.11% MND score over McSplitDAL. While this number lies well within the realm of statistical error, this behavior is constantly present across different runs, suggesting a minimal, yet real, causal advantage. Consequently, despite the small difference, we can declare McSplitDAL-SD the winner, and therefore all the future McSplitX+ variants will use the SD policy.

Joint vs Isolated Q-tables



(a) Rolling average ($W = 50$), normalized by McSplitDAL Joint, sorted by average.

Method	MND (%)	Average
McSplitDAL Isolated	**3.73%**	47.483
McSplitDAL Joint	0.0000%	45.650

(b) Test statistics. The MND is relative to McSplitDAL Joint.

Figure 6.5: Comparison of McSplitDAL Joint and McSplitDAL Isolated on SMALL

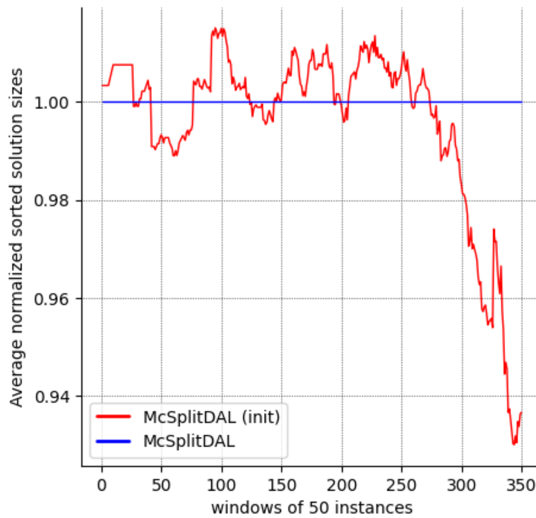
Unlike the SD policy, the McSplit Isolated does provide a more significant performance boost relative to McSplit Joint. As shown in 6.5, the isolated Q-tables variant of McSplitDAL achieves a convincing 3.73% difference over the competitor. This means that the differentiation of the learned policies between each of the two RL and DAL agents is considerably more beneficial than the small reduction in computational overhead brought by the joint version.

If the tested graph pair is large enough, the size of the pair Q-table $Sp(v, w)$ is $O(|G||H|)$, which could overflow the available memory. In the case of McSplitDAL Isolated, this memory consumption would be doubled, as each agent would have

its own Q-table. However, this does not increase the overall memory occupation complexity, thus relegating the use of the McSplitDAL Joint to the few rare cases of graph pairs for which $|G||H| < Max_memory < 2|G||H|$.

Consequently, all the future McSplitDAL+ variants in this thesis will use isolated Q-tables.

Initialization of Q-tables



(a) Rolling average ($W = 50$), normalized by McSplitDAL, sorted by average.

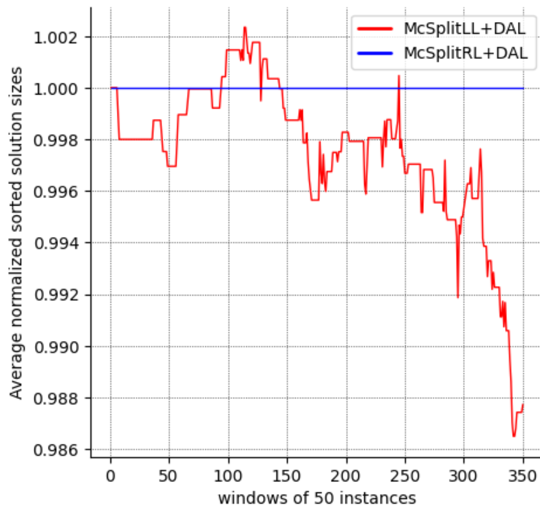
Method	MND (%)	Average
McSplitDAL (init)	-1.27%	46.625
McSplitDAL	**0.0000%**	47.483

(b) Test statistics. The MND is relative to McSplitDAL.

Figure 6.6: Comparison of McSplitDAL and McSplitDAL (init) on Small

6.6 shows that the initialization of the Q-tables with the heuristic sort order does not provide any significant advantage over the default initialization. Specifically, the results show a performance decrease of 1.27% relative to the zero-initialized McSplitDAL.

The gain plot in Figure 6.6a shows an interesting pattern, where the initialized version performs comparatively, if not slightly better, in the tougher test instances on the left of the plot, but the performance declines fast on easier instances, up to 6% worse in some windows. Overall, the results suggest that the initialization of the Q-tables with the heuristic sort order is not beneficial, especially on larger instances, therefore all the future McSplitDAL+ variants will use the zero-initialized Q-tables.



(a) Rolling average ($W = 50$), normalized by McSplitRL+DAL, sorted by average.

Method	MND (%)	Average
McSplitLL+DAL	-0.33%	46.625
McSplitRL+DAL	**0.0000%**	46.825

(b) Test statistics. The MND is relative to McSplitRL+DAL.

Figure 6.7: Comparison of McSplitLL+DAL and McSplitRL+DAL on SMALL

McSplitRL+DAL vs McSplitLL+DAL

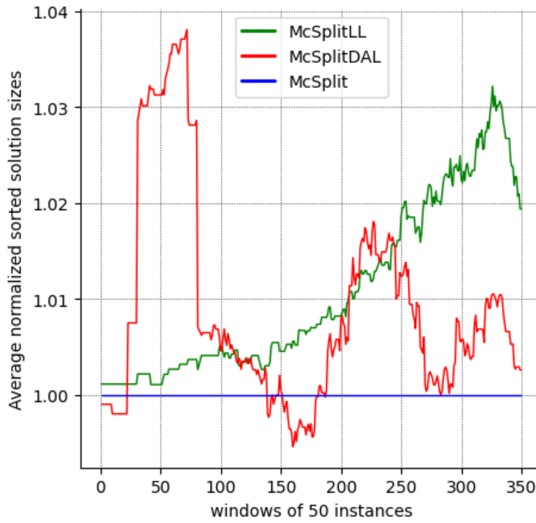
McSplitLL+DAL is the variant that uses two LL and DAL agents rather than two RL and DAL agents. 6.7 shows that the performance of McSplitLL+DAL is comparable to McSplitRL+DAL, with a slight performance decrease of 0.33% relative to the latter. The increased performance of LL relative to RL seems to not be significant enough to overcome the downside of the reduced agent differentiation, which leads to an increased fixation inside local minima.

The McSplitRL+DAL algorithm is the winner of this comparison, and therefore all the future McSplitDAL+ variants will use two RL and DAL agents.

Comparison of all McSplitX variants

At this phase of the testing process, we have identified the best-performing variant of McSplitDAL as the one using the SD policy, with zero-initialized isolated Q-tables for the pair rewards, and two RL and DAL agents. 6.8 compares the McSplitX variants McSplit, McSplitLL, and McSplitDAL. The results show that McSplitLL is the best-performing variant, with a performance increase of 0.97% relative to McSplit. McSplitDAL is the second-best variant, with a performance increase of 0.71% relative to McSplit.

These results seem to be in conflict with the data published by the authors of McSplitDAL, who reported a net performance increase over McSplitLL. We believe



(a) Rolling average ($W = 50$), normalized by McSplit, sorted by average.

Method	MND (%)	Average
McSplitDAL	0.71%	47.468
McSplit	0.0000%	47.193
McSplitLL	**0.97%**	47.767

(b) Test statistics. The MND is relative to McSplit.

Figure 6.8: Comparison of all McSplitX variants on SMALL

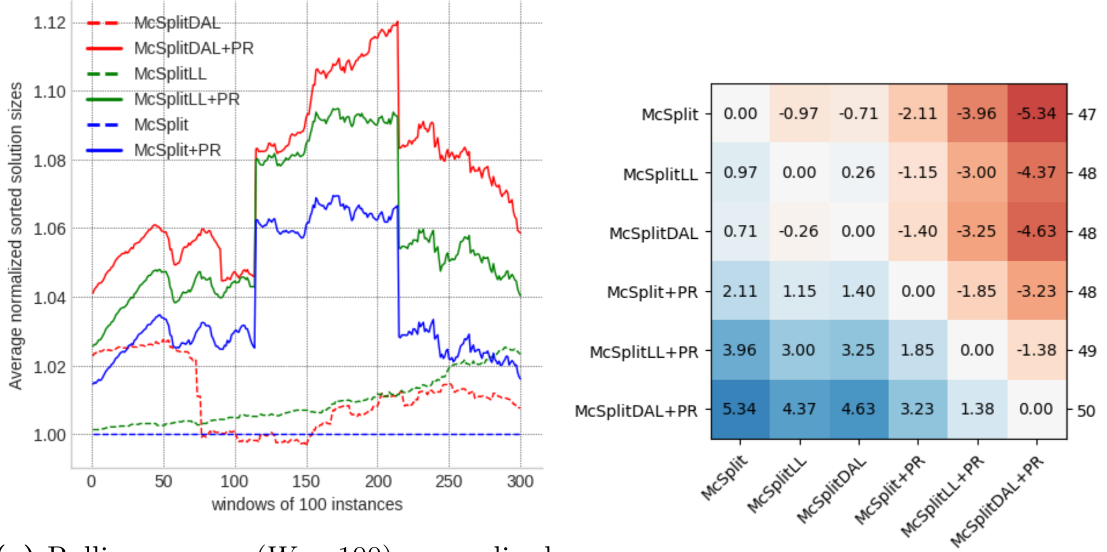
this difference is caused by our considerably different testing methodology, focused on finding the best solution before a strict timeout, rather than finding the optimal solution of many small MCS instances in the shortest time possible.

6.2.2 A first toe in the water with PageRank

Once the performance of the base McSplitX models has been measured as a baseline, we can start testing the static heuristics. We are first going to individually test the McSplitX variants with the PageRank heuristic, to get a first idea of the performance increase that can be achieved when changing the heuristic.

The results depicted in 6.9 present a comparison between all the McSplitX and McSplitX+PR variants. While the results for the McSplitX algorithms remain consistent with those displayed in 6.8, the current plot exhibits a slight variation due to the sorting of results based on the new average of all methods. Overall, it is evident that the McSplitX+PR variants outperform their respective McSplitX counterparts, demonstrating a performance increase of at least 2% across almost all test windows. Notably, the McSplitDAL+PR variant achieves a maximum average improvement of 5.34% relative to the base McSplit algorithm.

To better display the relative performance differences, the heatmap in Figure 6.9b presents itself as a matrix where each cell XY reports the mean normalized difference MND in percent of the variant Y (on the left axis) relative to the variant



(a) Rolling average ($W = 100$), normalized by McSplit, sorted by average

(b) MND heatmap (%)

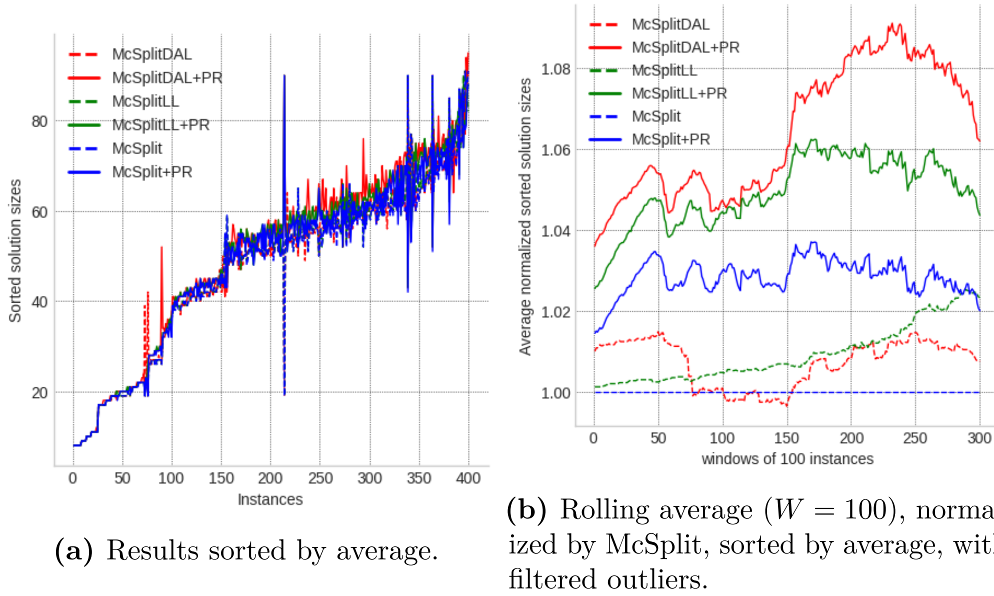
Figure 6.9: Comparison of all McSplitX+PR variants on SMALL

X (on the bottom axis). The diagonal shows the differences of all methods relative to themselves, and therefore is always 0%. The numbers to the right of each row are the average solution sizes of each Y method across all tests.

The gain plot in Figure 6.9a seems to have an anomaly, as all the McSplitX+PR variants seem to have a strong performance increase in 100 windows. By examining the same graph without the rolling average and normalization (Figure 6.10a), it becomes apparent that these exceptional results are attributable to a few prominent outliers. For instance, McSplitX performs remarkably poorly on instance 213, whereas McSplitX+PR achieves a significantly favorable outcome. Consequently, the resulting difference highlights an approximate 400% performance difference. While this particular data point is not representative of the overall algorithmic performance, it is not considered an outlier in the strict sense, as it is not a result of measurement error. Hence, it does hold meaningful implications.

However, to maintain fairness, Figure 6.10b displays the same graph with the outliers filtered out. The performance increase of McSplitX+PR remains largely unchanged, with McSplitDAL+PR persisting as the best-performing variant.

Moreover, the line plot reveals another intriguing phenomenon. When employing the degree heuristic, McSplitLL slightly outperforms McSplitDAL, yet McSplit-DAL+PR clearly outperforms McSplitLL+PR. This observation suggests that the PageRank heuristic compensates for the performance decrement of the DAL policy.



(a) Results sorted by average.

(b) Rolling average ($W = 100$), normalized by McSplit, sorted by average, with filtered outliers.

Figure 6.10: Analysis of outliers in McSplitX+PR variants on SMALL

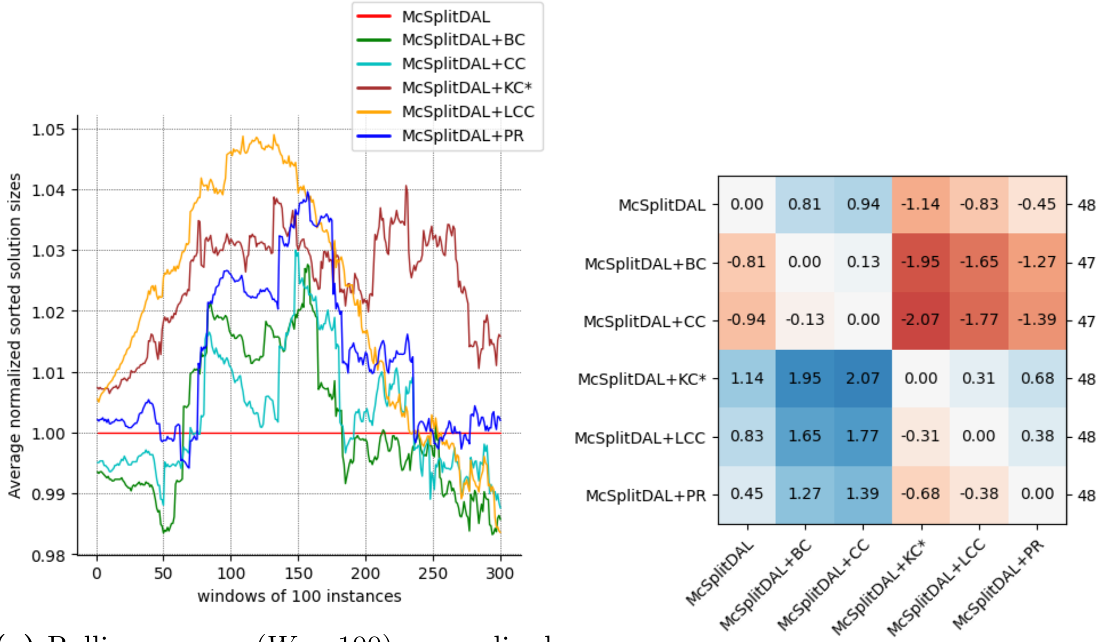
Such behavior can be attributed to the more intricate reward mechanism employed by DAL, which benefits from a sophisticated heuristic that guides the algorithm towards more favorable branches during the initial stages of recursion. Analogously, comparable effects can be observed in the other McSplitX+ variants, leading to the conclusion that McSplitDAL represents the superior variant among the McSplitX approaches when considering alternative sort orders. Consequently, moving forward, we will solely focus on presenting the results of the McSplitDAL+ variants.

6.2.3 Comparison of the static heuristics

Following the selection of McSplitDAL+ as the best-performing McSplitX+ family variant, we proceed to test all the static heuristics. As a reminder, we are going to compare the following node sort orders:

- *Degree*: The number of edges in the graph that are adjacent to the vertex.
- *PageRank (PR)*: Connections to high profile vertices
- *Betweenness Centrality (BC)*: Number of shortest paths that traverse the vertex
- *Closeness Centrality (CC)*: Average length of the shortest paths from the vertex

- *Katz Centrality* (KC*)*: Count of neighbors weighted by shortest distance
- *Local Clustering Coefficient (LCC)*: Cliqueness of the neighborhood



(a) Rolling average ($W = 100$), normalized by McSplitDAL, sorted by average

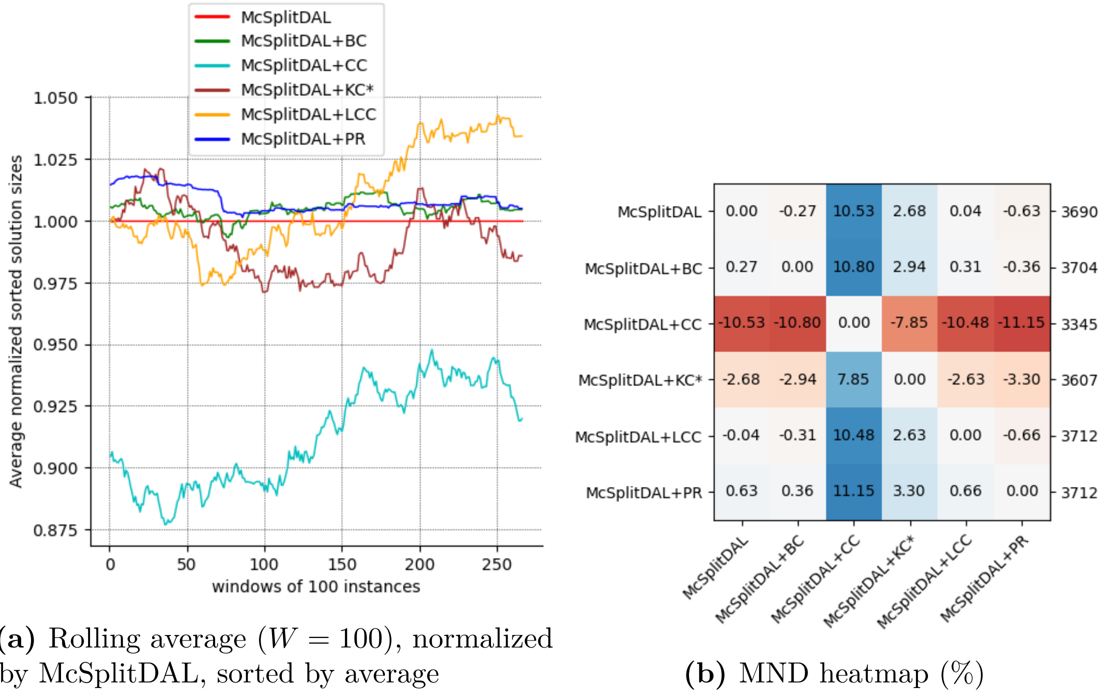
(b) MND heatmap (%)

Figure 6.11: Comparison of all McSplitDAL+ variants on SMALL

In Figure 6.11, the results of the McSplitDAL+ variants are depicted. The gain plot displayed in Figure 6.11a reveals that there is no clear winner across all test windows. On the left side of the graph, the Local Clustering Coefficient heuristic emerges as the front runner for the most complex problem instances. However, Katz Centrality* exhibits a distinct advantage on the simpler problem instances and performs comparably well across the entire test suite. In terms of average performance, Katz Centrality* proves to be the best-performing heuristic, demonstrating a 1.14% MND improvement over McSplitDAL. The Local Clustering Coefficient heuristic follows as the second best-performing approach, exhibiting a 0.83% gain.

Furthermore, the line plot unveils similarities between different heuristics. Both the Betweenness Centrality and Closeness Centrality algorithms follow comparable trends, as they are both rooted in the concept of shortest paths, with Betweenness Centrality performing slightly better. Similarly, PageRank, which is closely related to Eigenvector Centrality, exhibits a similar pattern but with marginally higher scores. This suggests that these heuristics encounter similar difficulty or ease in

classifying vertices based on similar criteria. On the other hand, the Local Clustering Coefficient generates a slightly different pattern, while Katz Centrality* stands out as the most distinct, with a clear advantage on the simpler problem instances. Consequently, these observations imply that on distinct datasets featuring graphs from diverse and specific domains, the choice of heuristic may need to be evaluated as a hyperparameter on a case-by-case basis.



(a) Rolling average ($W = 100$), normalized by McSplitDAL, sorted by average

(b) MND heatmap (%)

Figure 6.12: Comparison of all McSplitDAL+ variants on LARGE

Figure 6.12 presents the outcomes of the McSplitDAL+ variants on the LARGE dataset. The observed behavior diverges from that observed on the SMALL dataset. Katz Centrality*, which demonstrated competence in solving smaller and simpler instances, encounters challenges when confronted with larger graphs, resulting in inferior performance even compared to the default degree sort order. Moreover, the Closeness Centrality proves to be ill-suited for larger graphs, possibly due to its high algorithmic complexity, exhibiting a significant 10.53% decrease in average gain against McSplitDAL. On the other hand, the Local Clustering Coefficient continues to outperform other heuristics on easier or larger instances but experiences a substantial decline in performance at the opposite end of the spectrum. The Betweenness Centrality and PageRank heuristics maintain relatively consistent and comparable performance across the entire test suite. Notably, the PageRank heuristic emerges as the top-performing approach on the LARGE dataset, showcasing

a 0.63% increase over McSplitDAL.

Based on these findings, it can be concluded that Katz Centrality* is better suited for smaller and simpler problems, while the Local Clustering Coefficient performs well on medium-sized instances. On the other hand, Betweenness Centrality and PageRank prove to be versatile heuristics, displaying competitive performance on both small and large graphs. Specifically, PageRank consistently produces larger subgraphs than the current most efficient MCS solver, McSplitDAL, on both datasets.

These results indicate that McSplitDAL+PR represents the new state-of-the-art ground-truth solver for the MCS problems across the entire test suite. A comprehensive discussion of these findings can be found in our paper *A Web Scraping Algorithm to Improve the Computation of the Maximum Common Subgraph* [41], which was written as part of the thesis work.

6.3 Multi-Threading architectures

In this section, we present the results of the multithreaded implementations described in Chapter 4: McSplit MultiBranch and McSplit Branch Sharing. The initial testing is carried out on the LARGE-FINETUNING dataset. This choice is motivated by the dataset's larger graph sizes, which result in a greater search space. Such a setup is particularly conducive to uncovering the potential benefits of a parallel exploration and highlighting the distinctions between a focused approach and a broader one.

6.3.1 McSplit MultiBranch (MB)

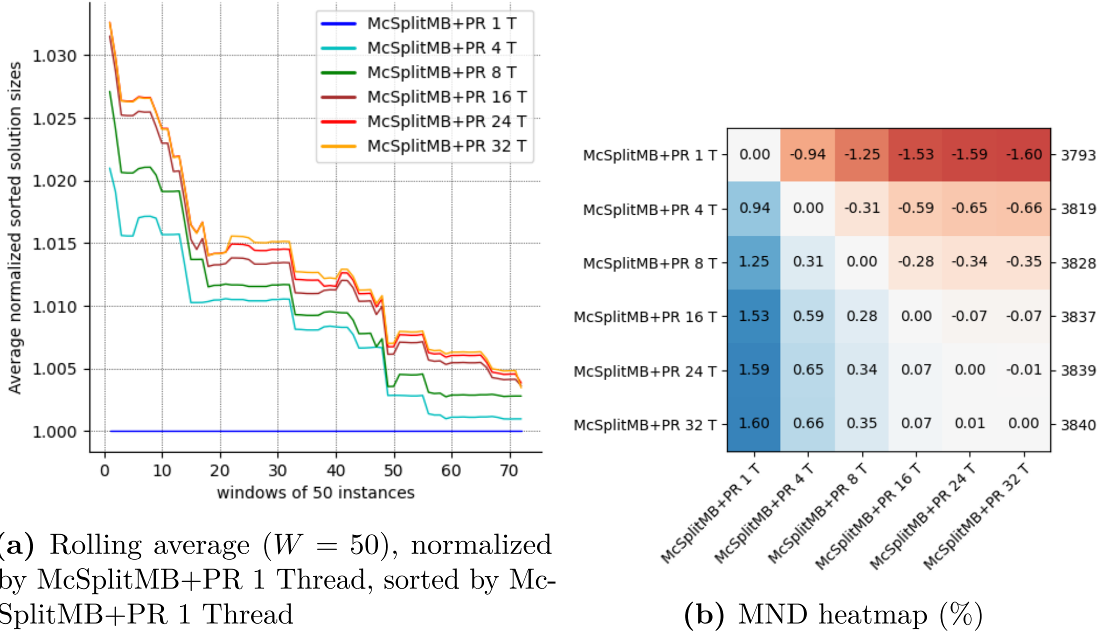
McSplitMB is a parallelized version of McSplit that utilizes a straightforward BFS strategy. The algorithm incorporates two hyperparameters: the number of threads, denoted as $n_{threads}$, and the *depth* at which the threads commence their independent exploration of their assigned branch. These parameters are highly dependent on the hardware specifications of the machine executing the algorithm, making it difficult to determine an optimal value universally. For our experiments, we employed a machine equipped with 10 cores and 20 hardware threads, and based on internal testing, we selected a depth value of 4. Additionally, the algorithm was executed on all the proposed heuristics to detect eventual differences in how they behave in a multi-threaded environment.

Thread count

The initial experiment conducted focuses on examining the impact of the number of threads on the performance of McSplitMB. The specific number of threads used is dependent on the hardware configuration and not the focus of the analysis, however, the resulting plot (Figure 6.13a) reveals an intriguing observation. The data is normalized relative to the performance of McSplitMB+PR 1 Thread, which uses the McSplitMB architecture, but it operates sequentially on a single thread. The test windows are sorted based on the outcomes of the 1 Thread version, therefore the test windows on the left represent those MCS problems that are comparatively more difficult to solve for the non-parallel algorithms.

We can see that all the other versions running on more than one thread share a very similar pattern relative to the sequential version. They can solve the easy tasks in a comparable amount of time, but they are significantly faster on those instances that 1 T struggles with. This finding suggests that when the sequential algorithm yields a good solution, that solution is not easily improved even by incrementing the number of threads. However, in harder instances where the 1 Thread version is unable to find a good solution, the introduction of additional

threads provides an advantageous opportunity to explore alternative branches. Overall, this increases the likelihood of finding a larger subgraph, and increases the resiliency of the algorithm when the initial branch selection is unpromising.



(a) Rolling average ($W = 50$), normalized by McSplitMB+PR 1 Thread, sorted by McSplitMB+PR 1 Thread

(b) MND heatmap (%)

Figure 6.13: Comparison of McSplitMB+PR using a different number of threads on LARGE-FINETUNING

Additionally, the plot demonstrates that the performance gain is not linear with the number of threads. Although the parallelization overhead is anticipated to be minimal due to the independent branch exploration, the performance gains do not scale proportionally to the number of threads. This indicates that the chosen heuristic (PageRank, in this instance) is sufficiently effective in identifying promising solutions within the initial branches, and the subsequent threads, which primarily investigate the last branches in the sort order, seldom discover better solutions.

Furthermore, it also highlights the complexity of improving the MCS solutions. Up until this point, all performance gains have been in the order of a few percentage points, at maximum. The fact that visiting a search space 16 times larger than the sequential version only yields a 1.53% improvement is a testament to the difficulty of the problem. On our machine, the most favorable outcome is achieved with 32 threads. The MND difference amounts to 1.68% over the sequential version on 1 thread. While the improvement may appear modest in relative terms, it is important to restate that the dataset includes significantly larger graphs compared

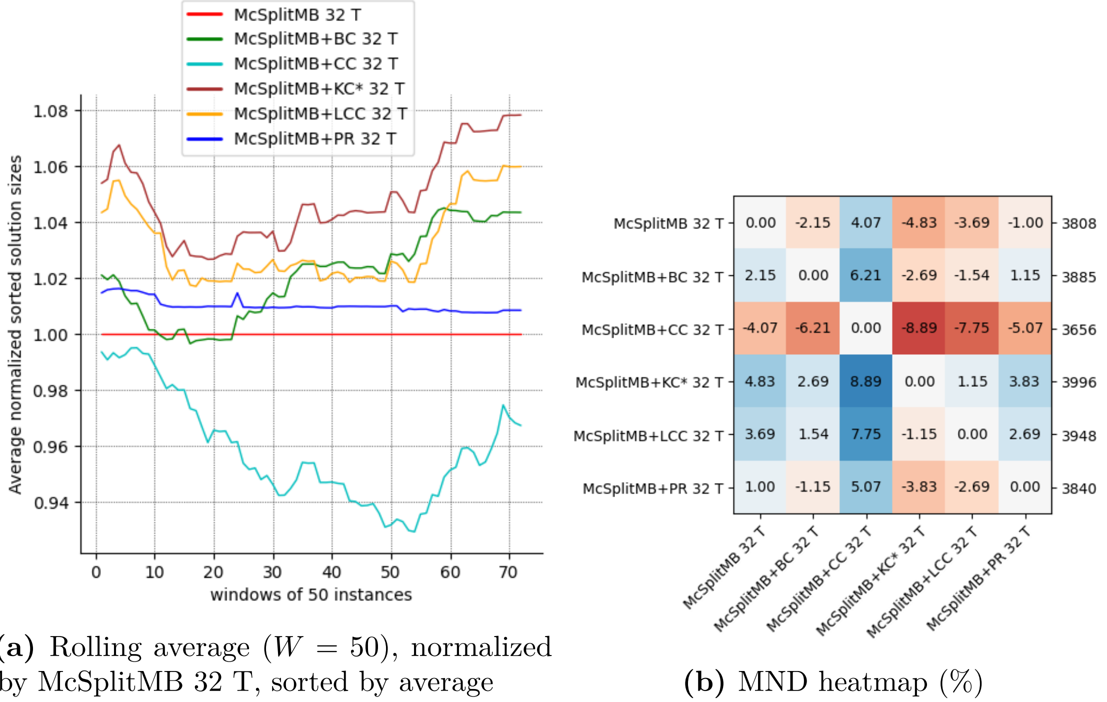
Small, with an average solution size of roughly 3800 nodes. Therefore, the 32-thread version produces subgraphs that are larger than the ones produced by the sequential version by tens of nodes. Considering the exponentially diminishing returns of the algorithm after the initial tree descent (6.1), this improvement holds substantial significance.

Lastly, we note that the best variant uses 32 threads, which is well beyond the number of available hardware threads. This phenomenon can partly be explained by the fact that the algorithm is not entirely CPU-bound, and the threads can spend a not insignificant amount of time waiting for the memory to be accessed. This can be especially true when the input graphs are very large, and the bidomain data structures cannot fit in the cache. In extreme cases, it is also possible that the required information is relegated to the SWAP partition of the computer, further increasing the access times. Therefore, the additional threads can be utilized to hide the memory latency, and the performance continues to improve even after the number of threads exceeds the number of hardware threads. However, another possible explanation is that the increased number of threads inherently forces the algorithm to explore more branches at once, effectively enforcing an even broader and shallower search.

Static Heuristics

The second set of experiments investigates the influence of the static heuristics on the performance of McSplitMB. For fairness, all tests were conducted with the same number of threads (32). The outcomes are depicted in Figure 6.14. The observed patterns mostly resemble the behavior exhibited by the heuristics in the sequential version of McSplitDAL (Figure 6.12a). McSplitMB+PR consistently demonstrates an improvement over McSplitMB, even considering the performance uplift observed by the multithreaded implementations on the most challenging instances, as observed in Figure 6.13a. Conversely, CC and LCC appear to gain even more substantial benefits from parallelization, particularly on the left portion of the graph. On the other side, BC exhibits significant gains on larger solution problems but loses its advantage on smaller ones. Finally, KC* seems to benefit the most from the parallelization of McSplitMB, getting back a convincing lead over the other heuristics, with a 4.83% difference relative to the default degree sort order. Despite its overall instability, KC* has the potential to be a very effective heuristic, if appropriate to the problem at hand.

In general, discernible patterns emerge, wherein each heuristic possesses its own strengths and weaknesses contingent upon the instance difficulty and parallelization. PR and BC consistently demonstrate stability across the test suite, establishing themselves as the most reliable heuristics.



(a) Rolling average ($W = 50$), normalized by McSplitMB 32 T, sorted by average

(b) MND heatmap (%)

Figure 6.14: Comparison of McSplitMB versions using 32 threads and different static heuristics on Large-Finetuning

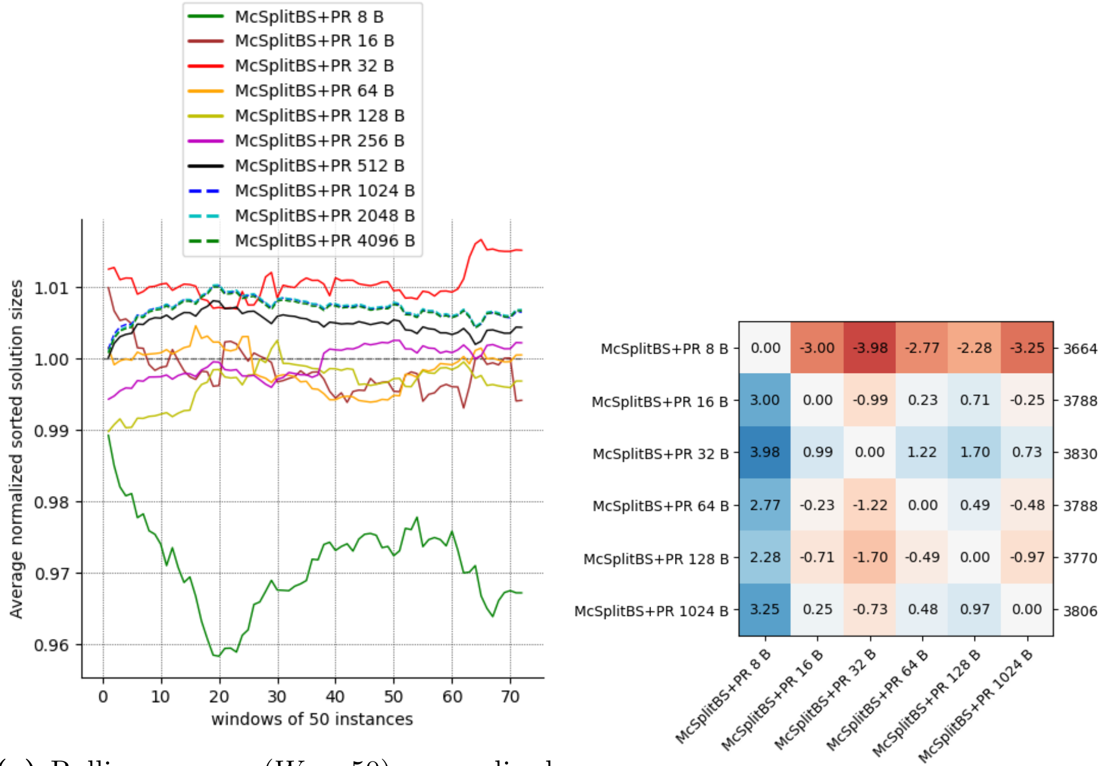
6.3.2 McSplit Branch Sharing (BS)

McSplitBS represents the second parallel variant of the McSplit algorithm, aiming for a more targeted approach to parallelization by enabling multiple threads to collaborate on the exploration of a single branch. Similar to the previous section, the experiments are conducted on the same machine, and the analysis follows a consistent structure. However, McSplitBS introduces an additional parameter, denoted as *block_size*, which necessitates separate testing and evaluation.

Block Size

Since there is no best candidate value for *block_size*, in this initial test McSplitBS is evaluated on a wide range of values. In keeping with the previous experiments, PageRank is selected as the static heuristic due to its consistent performance, and 32 threads are employed using the DAL policy. We recall that the *block_size* controls the maximum size of the local stack, dictating the extent to which each thread can independently explore a portion of the search tree before requiring synchronization with other threads. Thus, if the *block_size* is set to a value that is too small, it will lead to an excessive number of resynchronizations between

threads. If it is too large, the threads will not share enough, effectively acting as a poorly-optimized version of McSplitMB.



(a) Rolling average ($W = 50$), normalized and sorted by average

(b) MND heatmap (%)

Figure 6.15: Comparison of McSplitBS versions using PageRank, 32 threads, and different block sizes on Large-Finetuning

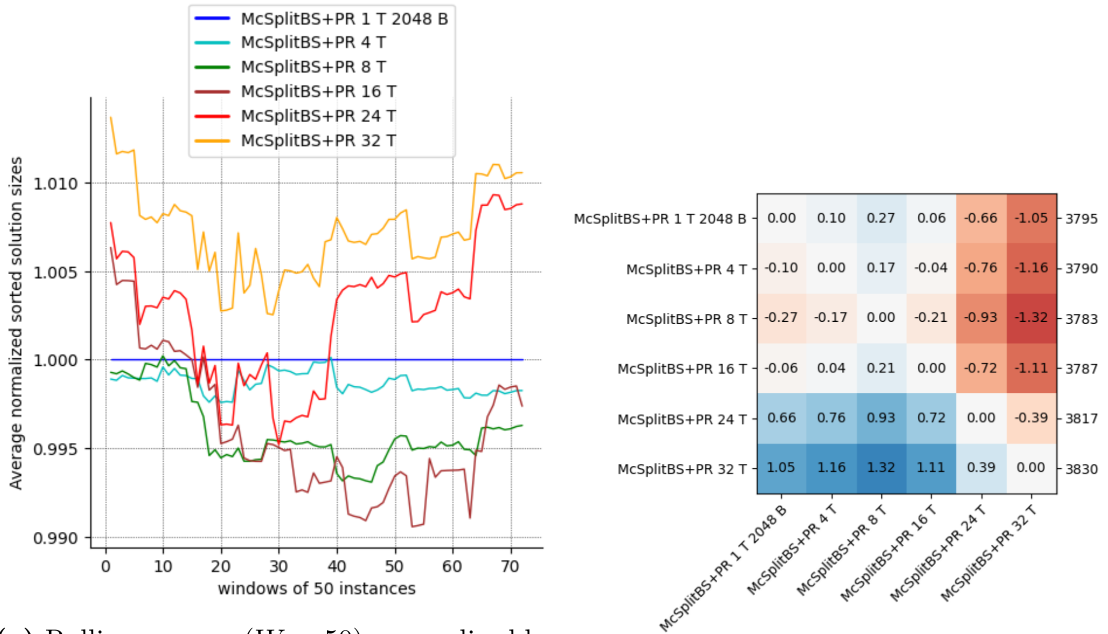
Some of the results are depicted in Figure 6.15, while a complete version of the difference heatmap is available in Appendix A. Since there is no baseline to compare against, all results are normalized based on the average solution size. Among the tested values, 32 emerges as the clear winner, with a difference of 1% over most competitors. Lower values of *block_size* exhibit a sharp drop in performance, with 8 falling notably below the average. On the other hand, high values are not as detrimental, maintaining an acceptable efficiency.

Interestingly the block sizes above 512 exhibit an almost identical pattern. The behavior is due to the fact that the in these cases the maximum size of the local stack is comparable to the length of the search tree, which entails that the first thread will likely share its branch only a few times during the first descent, and the other threads will likely never be assigned a branch that cannot fit entirely in the local stack. Consequently, the only data on the global stack will be the nodes that

were pushed by the first thread during the first descent. During the progression of the algorithm, each thread will independently explore the entire branch assigned to it, then pop the last node from the global stack and continue the exploration on a new branch. This behavior presents itself after a critical value of *block_size* is reached, and further increasing this parameter does not affect the performance. Furthermore, by allowing the threads to explore entire branches independently, the algorithm is greatly reducing the number of conflicts and the parallelization overhead, thus justifying the performance gains observed in the charts.

However, the data suggest that the optimal value for *block_size* is 32, which represents an intermediate search strategy: not as broad as McSplitMB, but not as strongly focused as the larger block sizes. Consequently, 32 is now confirmed as the default setting for the next experiments.

Thread Count



(a) Rolling average ($W = 50$), normalized by McSplitBS 1 Thread and sorted by average

	McSplitBS+PR 1 T 2048 B	McSplitBS+PR 4 T	McSplitBS+PR 8 T	McSplitBS+PR 16 T	McSplitBS+PR 24 T	McSplitBS+PR 32 T	
McSplitBS+PR 1 T 2048 B	0.00	0.10	0.27	0.06	-0.66	-1.05	3795
McSplitBS+PR 4 T	-0.10	0.00	0.17	-0.04	-0.76	-1.16	3790
McSplitBS+PR 8 T	-0.27	-0.17	0.00	-0.21	-0.93	-1.32	3783
McSplitBS+PR 16 T	-0.06	0.04	0.21	0.00	-0.72	-1.11	3787
McSplitBS+PR 24 T	0.66	0.76	0.93	0.72	0.00	-0.39	3817
McSplitBS+PR 32 T	1.05	1.16	1.32	1.11	0.39	0.00	3830

(b) MND heatmap (%)

Figure 6.16: Comparison of McSplitBS versions using PageRank, 32 block size, and different thread counts on LARGE-FINETUNING

The performance of McSplitBS with different thread counts is shown in Figure 6.16. The baseline is represented by the algorithm running on a single thread. Since this version does not gain any practical benefit from the use of the concurrent

architecture, it is run with a block size of 2048 to minimize the time spent on the *Step* migrations from the local to global stack, while the other parallel tests are conducted considering $block_size = 32$.

The results indicate that lower thread counts experience increased overhead, resulting in poorer performance compared to the baseline. However, the 24 and 32 threads versions are able to overcome this disadvantage through parallelization. Similar to McSplitMB, the 32 threads version of McSplitBS exhibits the best performance, with a 1.05% difference relative to the baseline.

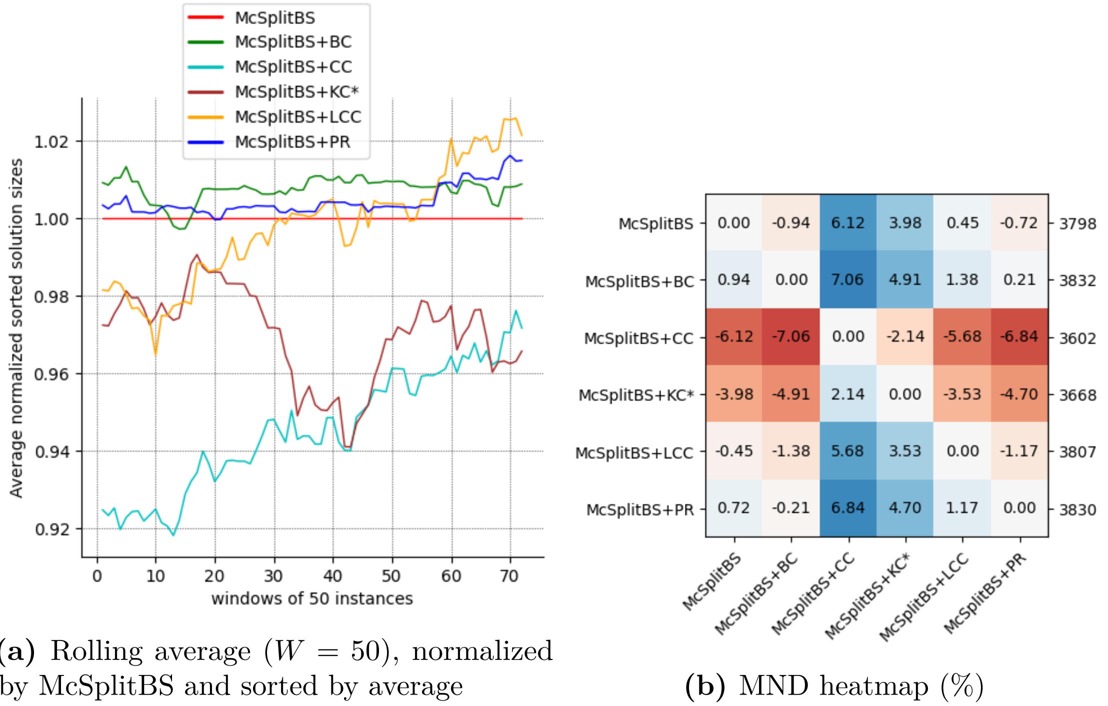
As noted for McSplitMB, it is interesting to note the noticeable performance difference between the 24 and 32 threads versions of McSplitBS, considering that the testing hardware only has 20 logical cores. This means that the threads cannot be allocated on the processor all at once, but they will have to be scheduled by the operating system. The observed performance difference is likely due to the fact that the 32 threads version is able to perform a wider exploration, as there are more agents that will be able to pick up the larger branches identified at the start of the algorithm. Conversely, it is more likely that the exploration of a branch will repeatedly be interrupted by the operating system, which will have to schedule other threads on the same core. Regardless, the data suggests that this is not a significant issue compared to the benefits of the wider exploration.

Moreover, the overall performance delta among all the considered thread counts is at most 1.32%, which is a relatively small difference. This indicates that the algorithm is not able to scale well with the number of threads, due to the considerable overhead introduced by the concurrent access to the global stack. Furthermore, McSplitBS incurs a significant algorithmic cost to save and manage the branching contexts inside the *Step* variable. This operation is necessary to replicate the recursive behavior of the original McSplit algorithm, which stores the same information on the call stack, and it cannot be easily optimized without a complete redesign of the algorithm.

Static Heuristics

As observed in the previous sections, each static heuristic aims to optimize the search for specific types of patterns within particular instances of the MCS problem. However, their performance is also influenced by the search strategy employed by the tested algorithm. Given the more complex nature of McSplitBS compared to both McSplit and McSplitMB, it is possible that the optimal static heuristic for McSplitBS differs from the one used in the previous experiments.

The results presented in Figure 6.17 provide some support for this hypothesis. CC, PR, and BC are largely unaffected by the change in algorithm, with BC being the most favorable choice. However, the performance of Katz Centrality* is notably poorer, indicating that this sort order may be only suitable for simpler



(a) Rolling average ($W = 50$), normalized by McSplitBS and sorted by average

(b) MND heatmap (%)

Figure 6.17: Comparison of McSplitBS versions using 32 threads, a block size of 32, and different static heuristics on LARGE-FINETUNING

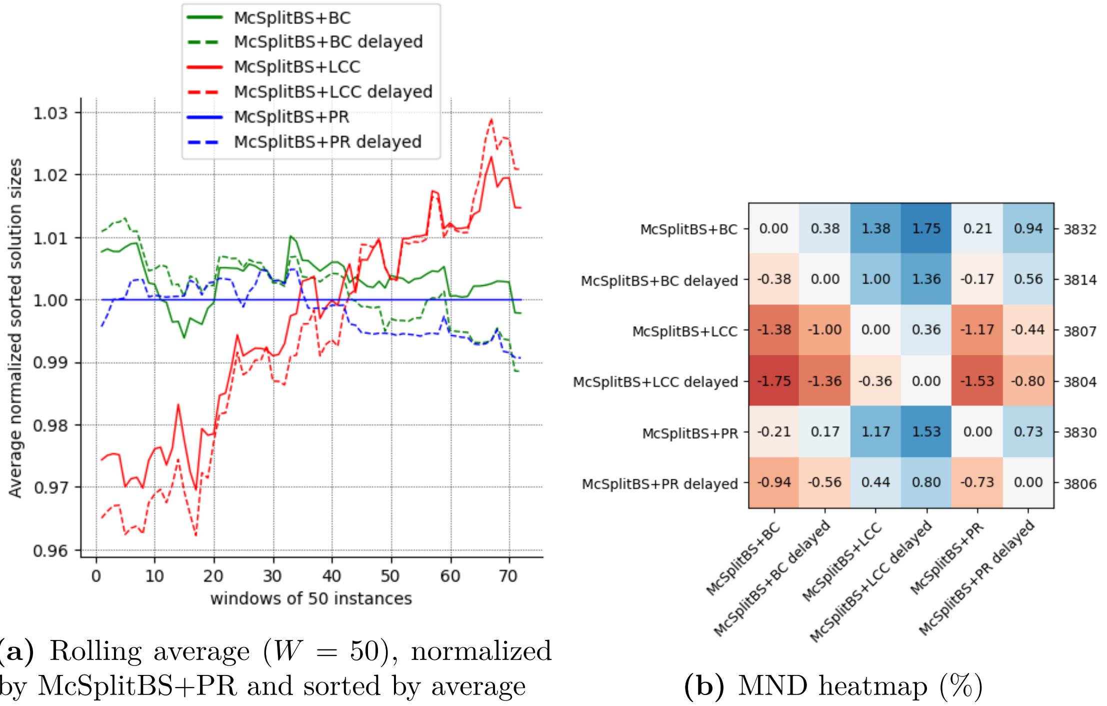
problems and parallelization strategies. Conversely, the Local Clustering Coefficient (LCC) exhibits a similar performance pattern to its sequential counterpart, McSplitDAL+LCC, and emerges as the leading heuristic for problems with larger solutions, but with unsatisfactory results in the smaller test windows.

Delayed Sharing

Delayed Sharing refers to the technique of preventing the threads from pushing any step on the global stack until the first pruning event occurs, regardless of the block size. This is intended to control the depth of the first branch sharing and force a more focused search in the first explored branches.

To not oversaturate the plot, 6.18 reports the results using only the most stable heuristics: PR, BC, and LCC. The full comparison charts are available in Appendix A. The delay mechanism seems to overall produce a negligible, if not harmful, effect, reducing the performance by up to 0.73% relative to the respective non-delayed algorithm. For this reason, the delayed sharing will not be considered in the following experiments.

However, it is worth noting that this mechanism does achieve promising results



(a) Rolling average ($W = 50$), normalized by McSplitBS+PR and sorted by average

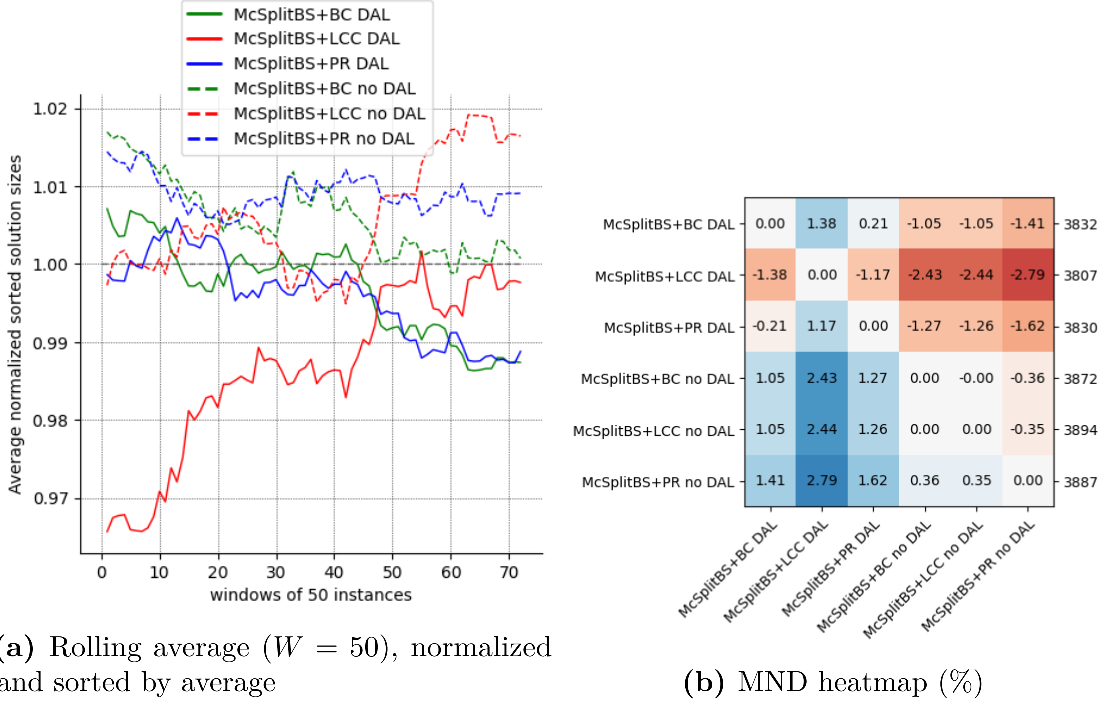
(b) MND heatmap (%)

Figure 6.18: Comparison of McSplitBS versions with or without delayed sharing using 32 threads, a block size of 32, and different heuristics on Large-Finetuning

in larger instances when using the LCC heuristic, which is the leader in this region of the chart. This suggests that the heuristics may be more capable of navigating larger search spaces, and only in this situation the delayed sharing might be beneficial. This hypothesis will have to be confirmed in future research with considerably larger graph pairs.

Is Reinforcement Learning effective in McSplitBS?

In McSplitBS, each thread undergoes multiple branch switches during the search process. This implies that the Q-table rewards employed in the DAL policy are learned and applied to different branches at different times. This could introduce noise during the branching decisions, as the learned RL context may have been directly transferred from another branch and immediately applied to the current one. Preliminary testing suggested that better performance could be achieved when the threads shared a common Q-table instead of utilizing local ones. This approach reduces the overfitting of a specific branch in favor of a more general policy. Conversely, it also reduces the effectiveness of the reward decay introduced by DAL, and it introduces additional overhead due to the necessary synchronization



(a) Rolling average ($W = 50$), normalized and sorted by average

(b) MND heatmap (%)

Figure 6.19: Comparison of McSplitBS versions with and without DAL policy, using 32 threads, a block size of 32, and different heuristics on LARGE-FINETUNING

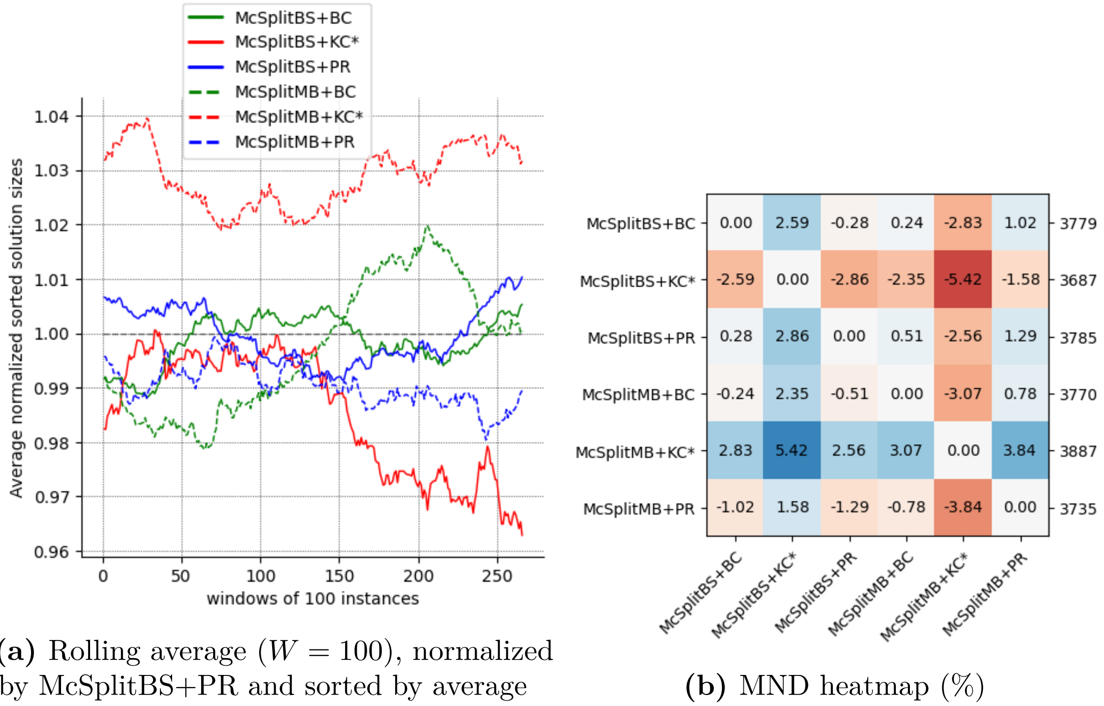
operation to guarantee mutual exclusion access to the Q-table.

Therefore, 6.19 reports the results of a test meant to evaluate whether the DAL strategy is still advantageous in McSplitBS. The findings indicate that this is not the case, with the non-DAL version of McSplitBS exhibiting a better performance in all the most stable heuristics PR, BC, and LCC. The results on the other heuristics are available in Appendix A. Consequently, in the following comparisons McSplitBS will be implicitly assumed to be the non-DAL version.

6.3.3 Conclusions on Multi-Threaded McSplit

Collected the results of the previous experiments, in 6.20 we compare the performance of the best performing versions of McSplitMB and McSplitBS. For clarity, the chart only shows a subset of the considered heuristics, namely PR, BC, and KC*. The complete results are available in Appendix A. For improved statistical precision, the tests are run on the entire LARGE dataset, using 32 threads and a block size of 32.

The results indicate that McSplitMB is overall the most efficient algorithm,



(a) Rolling average ($W = 100$), normalized by McSplitBS+PR and sorted by average

(b) MND heatmap (%)

Figure 6.20: Comparison of the best performing versions of McSplitMB and McSplitBS using 32 threads, a block size of 32, and different heuristics on LARGE

especially when combined with its most favorable heuristics KC* and LCC. Interestingly, this pattern is not consistent across all sort orders, as McSplitBS+PR exhibits a better performance than McSplitMB+PR. Similar behavior is observed for degree-based variants as well.

As already stated, the performance of McSplitBS is heavily influenced by the choice of the static heuristic. Consequently, it is essential to select the most suitable one for the specific problem at hand. However, on our specific test set, PageRank and the original node-degree are the only sort orders that benefit from using McSplitBS over McSplitMB. In all other cases, McSplitBS is either slower or exhibits a negligible performance gain.

This suggests that the more focused approach of McSplitBS cannot outmatch the broader exploration of the search space produced by McSplitMB, finding advantageous applications only in specific scenarios. Particularly, McSplitBS finds success with an effective and stable heuristic such as PageRank because it is able to find large solutions in the first branches of the search. If the heuristic is not as effective, allocating more resources to other branches will more likely lead to more satisfactory results. However, it is also possible that the performance of McSplitBS could be improved by further tuning the width of the search and the efficiency of

the parallel architecture.

6.4 GNN models

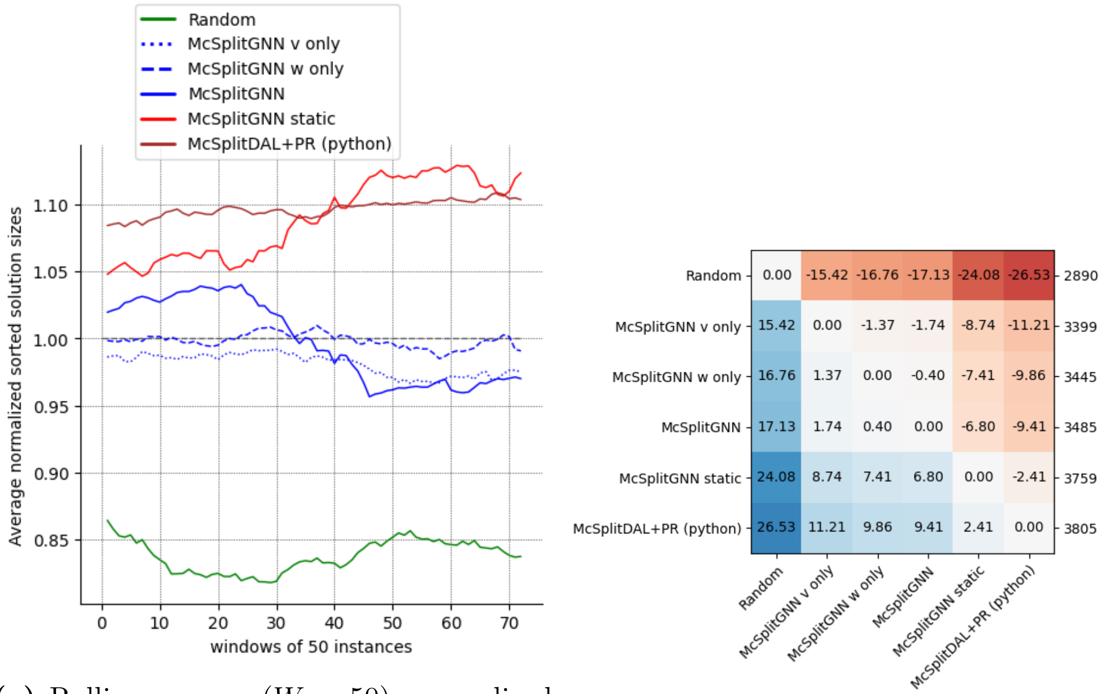
This section is reserved for the analysis of the results of all GNN-based models. While naturally interested in identifying the best-performing approach, the main goal of this exploration is to understand the behavior of the GNNs and the impact of the different parameters on the results. While all the considered models are written in Python to make use of the popular PyTorch machine learning libraries, the McSplit code and the testing methodology are equivalent to the previous implementations in C++. The following subsections will present the individual results of McSplitGNN, McSplit DiffGNN, and their variants, while the last subsection will present a comparison of all methods and the conclusions that can be drawn from them.

6.4.1 McSplitGNN

McSplitGNN is the simplest of the models discussed in Chapter 5, relaying on a GCN to produce node-level embeddings which are then converted into scalar scores by a MLP. The model has been trained on the branching context information extracted from the McSplit exploration of 10 graphs from the Large dataset. Each of these explorations produces from 100 thousand to 50 million branching contexts, depending on the graph size. To preserve the asymmetric nature of McSplit, two separate models have been trained, one on the branching decisions used to select v, and the other on the decisions used to select w. These two models will be then used independently to score the nodes in the branching context of a new graph pair. The GNN architecture has to be trained on bidomains of variable size, so the batch size must be set to 1. The low batch size means that the gradient descent is performed on each sample, so training was performed in only 3 epochs, with a learning rate of 0.005.

McSplitGNN was tested on the Large-Finetuning dataset. In 6.21 the following variants are compared:

- McSplitGNN: the model described above.
- McSplitGNN v only: the GNN model is used to score v, while w is selected using the DAL policy with the PageRank heuristic.
- McSplitGNN w only: the GNN model is used to score w, while v is selected using the DAL policy with the PageRank heuristic.
- McSplitGNN static: at the start of the algorithm the GNN model is run on the entire graphs G and H to generate node scores and compute a static sort order. Then the branching decisions are performed using the DAL policy with this new GNN-based heuristic.



(a) Rolling average ($W = 50$), normalized and sorted by average

(b) MND heatmap (%)

Figure 6.21: Comparison of all McSplitGNN and McSplitGNN static on Large-Finetuning

- Random: a very simple McSplit variant, implemented in Python, that randomly selects v and w from the branching context. This variant is used as a baseline to compare the performance of the other models.
- McSplitDAL+PR: the McSplitDAL algorithm with PageRank heuristic, which serves as a baseline for the performance of the McSplitGNN variants. For fairness, this algorithm is also implemented in Python instead of the original C++.

All the considered models perform considerably better than the random algorithm, thus confirming that the GNN is able to learn some useful information from the branching context. McSplitGNN obtains a higher gain over the average when the Neural Network is selecting both vertices, rather than just one of them. This discrepancy in the v-only and w-only versions is likely caused by a mismatch in the node priority order on the two sides of the McSplit algorithm, consequently rendering the resulting vertex matching less effective. However, this effect seems to have a moderately positive effect on the larger MCS instances to the right of the plot.

The most interesting observation is related to the static version of McSplitGNN, which is able to greatly outperform the other variants across all windows. This Transfer Learning strategy uses the optimization knowledge gathered from dynamic branch decisions made on small bidomains, and it adapts it for the computation of one-shot static scores on the entire graphs. The presented results are a clear representation of the innate advantage that static heuristics have over dynamic ones. During each branching event, McSplitGNN has to convert the bidomain node lists to two independent graph objects, which alone requires considerable CPU time, and then run the two GNN models to score the nodes. This overhead is completely avoided by the static version, which is able to compute the node scores once and then use them for the entire duration of the algorithm. Furthermore, the static version has the ulterior advantage of being able to use the DAL policy and the power of Reinforcement Learning.

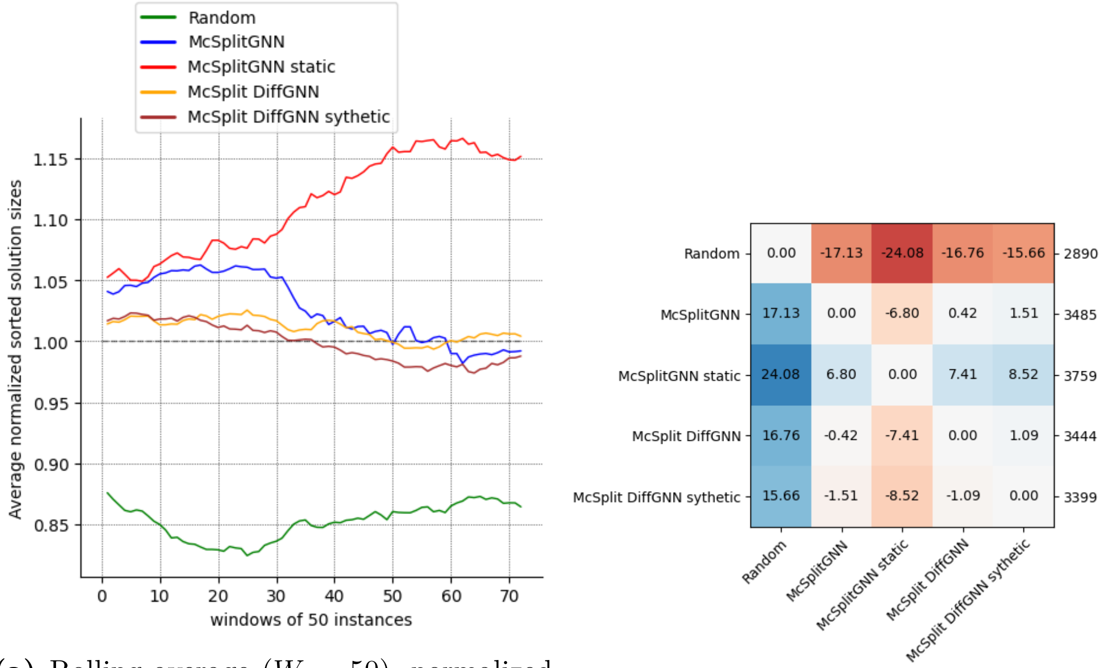
This Transfer Learning approach used to adapt dynamic decision-making policies to static heuristics shows results that are comparable with McSplitDAL+PR, the new state-of-the-art variant of McSplit, with an MND score difference of 2.41%. This is a promising result, which might be easily improved by adopting a learning strategy more suited to static heuristics.

Consequently, future machine learning models applied to the McSplit algorithm should target one of two distinct paths. The first is to develop a more complex model that is purposely built to produce an effective static sort order, which can then be used by the DAL policy. The second is to develop a model that is able to produce node scores in a very short amount of time, so that the dynamic version of the model can be used without incurring significant overhead. The potential advantage of the latter strategy is that the model could take advantage of the local branching context, which is not available to the static version. To keep track of the current and past states, the model would likely require a more complex architecture, possibly involving integration with a DQN (as in the case of GLSearch) or Recurrent Neural Networks.

6.4.2 McSplit DiffGNN

McSplit DiffGNN uses a GCN model with the aim of assigning similar embeddings to nodes v and w that should be matched. The training is done again on branching contexts extracted from the McSplit exploration of 10 graphs from the Large dataset. Since it uses a single model, the training was extended to 5 epochs.

The same model was then trained on the modified branching contexts extracted from a synthetic dataset, as discussed in Section 5.3.3. The dataset was created from Large graphs using a vertex probability $p_v = 10\%$ and an edge probability $p_e = 2\%$.



(a) Rolling average ($W = 50$), normalized and sorted by average

(b) MND heatmap (%)

Figure 6.22: Comparison of McSplit DiffGNN and DiffGNN synthetic on Large-Finetuning

6.22 compares the performance of the two implementations on the Large-Finetuning dataset. While the observed difference is not as significant compared with the other models, currently DiffGNN synthetic is not able to provide a significant advantage over the original model. The most likely explanation is that the synthetic graphs do not present the same level of complexity as the real-world graphs, and consequently the model is not able to learn a more effective representation of the branching context. However, the results are still promising, and future research might be able to improve the specifications of the generation of synthetic graphs to achieve more satisfactory results. Specifically, the synthetic dataset might be more effectively employed by using it as a step of a multi-stage curriculum learning process.

Both DiffGNN models are not able to confidently reach the solution sizes of the simpler McSplitGNN. However, this is not an unexpected result. These models only use the differential matching system for the selection of the vertex w, while the selection of v is still done using the DAL policy with the PageRank heuristic. Future research might be able to achieve higher performance by using the model to select entire graph pairs.

6.4.3 Conclusions on the GNN-based models

The architectures of the GNN-based models are extremely simple, considering the complexity of the problem under discussion, and the training is relatively fast, with a maximum time of a few hours. The obtained results are promising, as they show that some knowledge can be obtained from the node-level embeddings, but they also highlight the challenges that are still present in the field of GNN research applied to the MCS problem.

The main issue is that any dynamic GNN-based model is necessarily slower than a static heuristic since it requires the computation of the node embeddings at each branching event. The dynamic version is able to execute a fewer number of iterations, and therefore it can explore fewer branches, consequently it has to perform more accurate branching decisions to be able to reach a good solution in the first branches.

At this time, the data suggests that the path forward for future research is to focus on improving the performance of the static version of the model, up to the point of matching, or even surpassing, the performance of the other heuristics considered in this thesis. Once this goal is achieved, it is likely that the static approach might reach a maximum efficiency cap, considering that the scores cannot change and adapt to the developments of the search process. This learned knowledge could then be integrated with the branching contexts in a dynamic model which uses Recurrent Neural Networks or Reinforcement Learning techniques, with the aim to overcome the limitations that come from the slower search speed, and hopefully achieve a new state-of-the-art solver for the MCS problem.

6.5 Summary of Results

This chapter presented a comprehensive analysis of three different families of algorithms for the Maximum Common Subgraph (MCS) problem. The performance of various approaches within each family was evaluated, and the best-performing algorithms were identified through comparative testing. It is worth noting that other slight variations of these algorithms were developed and tested during the research, but their performance did not meet the desired criteria and no interesting conclusions could be extracted from these attempts, therefore they were not included in this thesis.

6.5.1 Qualitative comments of the results

Static heuristics The first family includes a set of implementations of McSplit-DAL, among which the best-performing version uses the SD swap strategy, coupled with zero-initialized isolated Q-tables for the RL and DAL policies. The algorithm is then further improved by using a collection of heuristics to reach a larger solution before a fixed timeout of 60 seconds. The behavior of the different sort orders is highly dependent on the domain of application and the solver used, but PageRank was selected as the most stable and overall best-performant variant.

Multi-threading The second family investigates multi-threading techniques through the McSplitMB and McSplitBS parallel implementations. The parameters for these algorithms were carefully selected based on comparative testing, and in the final experiments both algorithms were executed on 32 threads, with a block size of 32 and excluding delayed sharing in the case of McSplitBS. Both variants did not use the DAL policy. The results indicate that McSplitMB using Katz Centrality* is the fastest parallel algorithm for the MCS problem in specific applications. However, McSplitBS remains competitive when employing the more stable heuristic, PageRank. Consequently, McSplitMB+ can be tentatively considered the fastest parallel algorithm for the MCS problem, but McSplitBS+ might still constitute a valid and more stable alternative in some domains.

Graph Neural Networks The third and last family is related to the GNN-based models. Unlike the previous families, this group of comparisons was not necessarily aimed at finding the fastest solver, but rather to gain a deeper understanding of the challenges of applying a Machine Learning approach to the MCS problem, and hopefully build a knowledge base for future research. Our investigation concluded that a GNN model used to compute node scores as a static heuristic can more easily obtain satisfactory results without a significant increase in complexity. On the other hand, models that attempt to act as a dynamic decision agent require

a more articulated architecture capable of overcoming the drawbacks of having a lower search speed.

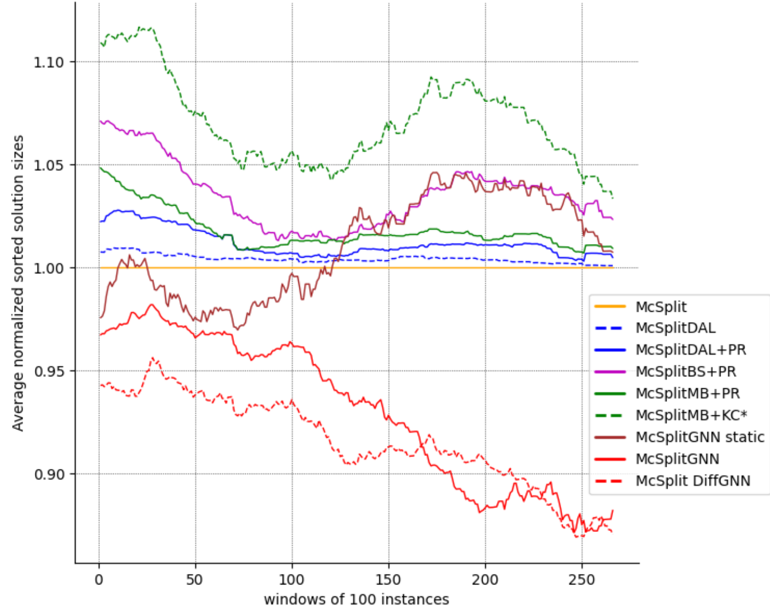
Final considerations on heuristics Considering the results observed in the first two categories, we can provide a description of the considered heuristics as follows:

- *PageRank (PR)*: This heuristic demonstrates high stability and consistently outperforms the original degree heuristic. Although it may not be the most efficient option when parallelized, it still yields satisfactory results. Overall, PageRank can be considered the most effective sort order in a general domain, and therefore McSplitDAL+PR represents the new state-of-the-art sequential MCS solver.
- *Betweenness Centrality (BC)*: Another stable heuristic, although slightly less so than PageRank. While it may be less effective than PageRank in sequential explorations, it gains a significant advantage when used in parallel.
- *Local Clustering Coefficient (LCC)*: A semi-stable heuristic that tends to perform better on larger instances. It benefits from the McSplitMB architecture but is not as effective in McSplitBS.
- *Katz Centrality* (KC*)*: A very unstable heuristic that can produce excellent results, particularly on smaller and simpler graphs, but it can also perform very poorly. It is largely the most effective heuristic when used in parallel with the simple parallelization strategy of McSplitMB, but it shows unacceptable performance in the more complex McSplitBS. While KC* can deliver solutions of outstanding quality, it should be carefully vetted to make sure it suits the target domain.
- *Closeness Centrality (CC)*: This heuristic regularly performs worse than any other, including the original degree. It is not recommended for any application.

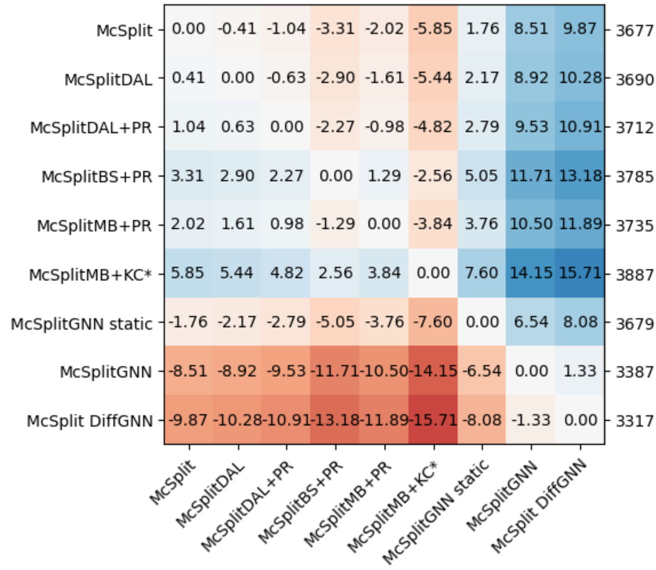
6.5.2 Quantitative comparison of the best algorithms

6.23 reports a comparison of the best models from each family on the LARGE dataset. Starting from the bottom, McSplitGNN and McSplit DiffGNN produce considerably smaller solutions than the other algorithms due to their relatively immature implementations.

Among the static heuristics used with the DAL policy, PageRank is the most stable, constantly delivering higher quality solutions than the default degree-based McSplitDAL, with a positive difference of 0.63%, and it outperforms the original McSplit by 1.04%.



(a) Rolling average ($W = 100$), normalized by McSplit and sorted by average



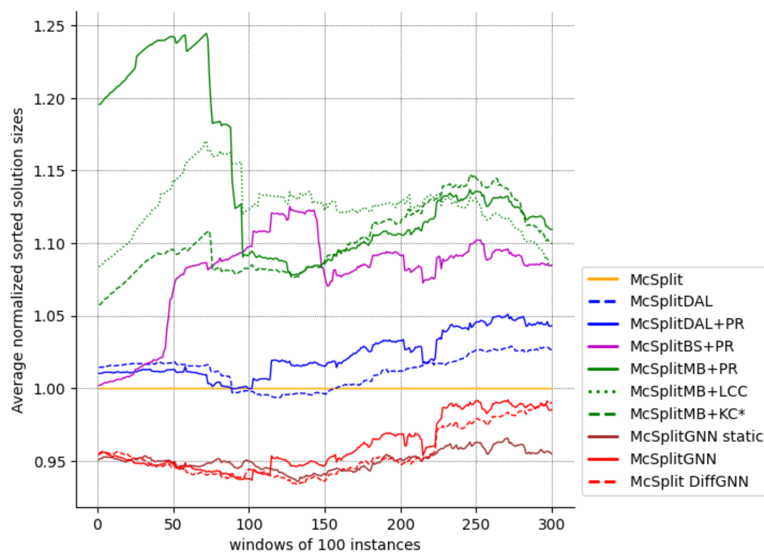
(b) MND heatmap (%)

Figure 6.23: Comparison of the best-performing algorithms for each family on LARGE

This difference is further increased by employing multiple threads. Using PageRank, McSplitBS is able to outperform McSplitMB by 1.29% and the sequential version McSplitDAL+PR by 2.27%. However, on the LARGE dataset McSplitMB is able to obtain considerably better results when using Katz Centrality*, with an overall difference of 5.44% over McSplitDAL. However, it is important to note that the results obtained by McSplitMB+KC* are highly unstable, and the same algorithm can perform very poorly on some other test sets. The data clearly highlights the importance of selecting the right heuristic for the target domain. Moreover, the numbers here reported are measured on the algorithms running on 32 threads, and different machines with other parallelization capabilities might obtain different results.

The static version of McSplitGNN has a negative score of 2.17% relative to McSplitDAL, but interestingly, the gain plot reveals that on larger instances it can outperform the most efficient sequential solver McSplitDAL+PR and it is comparable to the parallel algorithm McSplitBS+PR. Its performance is even more impressive considering that it is implemented in Python, which is a much slower language than C++.

On the SMALL dataset (6.24) the algorithms show a similar, but different, behavior. The GNN-based models are still the worst-performing, but by a smaller margin. This is likely not a merit of the GNNs, but rather a consequence of the fact that the SMALL dataset is composed of smaller graphs, which are easier to solve. Interestingly, the static version of McSplitGNN is not as effective anymore, showing similar performance to the other GNNs. This suggests that the learned policies are not effective on small graphs, likely due to the fact that the training set was composed of larger instances. McSplitMB confirms its superiority over McSplitBS, winning even when using the PageRank heuristic. Katz Centrality* is still the most effective sort order for easier instances, but it loses considerably in the left region of the plot. This is consistent with what was observed in Section 6.2.3. On this smaller dataset, the winning algorithm is McSplitMB+PR, closely followed by McSplitMB+LCC.



(a) Rolling average ($W = 100$), normalized by McSplit and sorted by average

	McSplit	McSplitDAL	McSplitDAL+PR	McSplitBS+PR	McSplitMB+PR	McSplitMB+LCC	McSplitMB+KC*	McSplitGNN static	McSplitGNN	McSplit DiffGNN	
McSplit	0.00	-1.08	-1.53	-5.63	-9.51	-8.72	-7.78	5.25	4.63	4.59	47
McSplitDAL	1.08	0.00	-0.45	-4.56	-8.49	-7.64	-6.71	6.33	5.70	5.67	48
McSplitDAL+PR	1.53	0.45	0.00	-4.12	-7.99	-7.20	-6.26	6.78	6.16	6.11	48
McSplitBS+PR	5.63	4.56	4.12	0.00	-3.85	-3.08	-2.13	10.86	10.24	10.20	51
McSplitMB+PR	9.51	8.49	7.99	3.85	0.00	0.85	1.75	14.72	14.10	14.06	52
McSplitMB+LCC	8.72	7.64	7.20	3.08	-0.85	0.00	0.94	13.93	13.31	13.29	52
McSplitMB+KC*	7.78	6.71	6.26	2.13	-1.75	-0.94	0.00	13.01	12.39	12.35	51
McSplitGNN static	-5.25	-6.33	-6.78	-10.86	-14.72	-13.93	-13.01	0.00	-0.62	-0.66	45
McSplitGNN	-4.63	-5.70	-6.16	-10.24	-14.10	-13.31	-12.39	0.62	0.00	-0.04	45
McSplit DiffGNN	-4.59	-5.67	-6.11	-10.20	-14.06	-13.29	-12.35	0.66	0.04	0.00	45

(b) MND heatmap (%)

Figure 6.24: Comparison of the best-performing algorithms for each family on SMALL

Chapter 7

Conclusion

The research work described in this thesis was aimed at improving the performance of the most effective ground-truth solver for the Maximum Common Subgraph (MCS) problem, McSplit. The research has been carried out in three main directions: the exploration of algorithmic optimizations and new static heuristics to prioritize the most promising areas of the search space in the improved McSplitDAL algorithm, the development of multi-threading techniques to fully leverage the computational power of modern multicore processors, and the application of Graph Neural Networks to guide the algorithm through a supervised approach.

One of the key challenges of the MCS problem lies in its algorithmic complexity. MCS is known to be NP-hard, making it unfeasible to find an optimal solution within limited time constraints. Without a reference target, it becomes challenging to assess the quality of the solutions found by the solver under scrutiny, or even to produce good-quality labels for Machine Learning models. Ultimately, the final objective of the thesis is to increase the time efficiency by finding the largest common subgraph possible between two graphs in the allocated time budget.

The research led to the formulation of a new algorithm, McSplitDAL+PR, which incorporates the PageRank algorithm, widely known for its application in the Google search engine, to approximate a best-first node selection policy. The algorithm produces statistically larger solutions compared to previous methods and now represents the new state-of-the-art solver for the MCS problem. This achievement warranted the publication of a paper titled *A web scraping algorithm to enhance maximum common subgraph computation* in the proceedings of the 2023 International Conference on Software Technologies (ICSOFT 2023).

However, alternative heuristics, based on Betweenness Centrality, Local Clustering Coefficients, and a modified Katz Centrality variant, proved to be particularly effective in specific use cases. Consequently, to achieve the highest level of performance, the selection of the most appropriate heuristic should be considered a hyperparameter to be fine-tuned for the specific domain of application.

Likewise, the two considered parallel architectures McSplit MultiBranch (MB) and McSplit Branch Sharing (BS) offer different levels of performance depending on the application and the adopted heuristic, but ultimately the MB variant demonstrated greater effectiveness and scalability in the majority of cases.

Finally, the Graph Neural Network (GNN) approach proved to be a promising path to explore, particularly when employed to generate a static vertex sort order for the McSplit algorithm. Through a Transfer Learning approach, our GNN model was able to generate node scores that ultimately led to a comparable performance relative to the other static heuristics, thereby paving the way for future advancements in this area.

The research has been conducted as a broad, but shallow, exploration of different approaches, all designed to address the MCS problem, and as such it has left many open questions and avenues for future research. While it is not feasible to list all possible improvements that could be explored, we focus on highlighting the most promising ones.

One particularly intriguing direction is the application of the GNN approach to construct a dynamic search policy which could lead to a significant improvement in the performance of the algorithm. The agent would rely on a Deep Q Network or on Recurrent Neural Networks to maintain knowledge of the current state, while employing a Graph Convolutional Network to explore the graphs. Furthermore, designing the model to leverage fixed-size data structures would enable optimal utilization of the computational capabilities offered by modern GPUs.

Regarding parallel processing, the Branch Sharing variant could benefit from the implementation of a more sophisticated thread management system. Such a system would create larger tasks and prioritize each compute unit based on their probability of discovering a solution. This approach would enhance the algorithm's ability to effectively leverage the computational power of the system while minimizing time wasted on tasks that are unlikely to yield helpful solutions.

The work presented in this thesis makes a concrete contribution to the research on the Maximum Common Subgraph problem. The findings and techniques developed in this study have the potential to positively impact other scientific domains and industries, including structural biology, computational chemistry, and software engineering. Furthermore, the wide array of algorithms presented in this study, along with the extensive range of experiments undertaken, provide helpful insights on the MCS problem and serve as a valuable resource for future researchers seeking to advance the state-of-the-art in the field.

Contributions

The thesis work is the result of an engaging and stimulating collaboration between me and my fellow colleague Salvatore Licata (Politecnico di Torino - candidate for Master's degree in Computer Engineering). The work was supervised by Prof. Stefano Quer (Politecnico di Torino) and Prof. Abolfazl Asudeh (University of Illinois at Chicago), as well as the Ph.D. candidates Andrea Calabrese and Lorenzo Cardone (Politecnico di Torino).

While most of the research has been conducted as a tightly collaborative effort between me and Salvatore Licata, I can summarize my major contributions as follows:

- **Chapter 3:** Exploration and implementation of some of the proposed McSplitDAL variants, including the SD policy and the comparison between McSplitLL+DAL and McSplitRL+DAL. Research and implementation of the static heuristics for McSplitDAL (in sequential or parallel versions).

- **Chapter 4:** Adaptation of the static heuristics on McSplitMB. Design and implementation of the McSplitBS architecture, including the delayed sharing variant.

- **Chapter 5:** Extensive analysis and reformatting of GLSearch to attempt to reproduce the results of the original paper, with a particular focus on the curriculum learning structure. Design and implementation of the McSplitGNN architecture, including its transfer learning application in the static version. Design and implementation of the model and the training pipeline of McSplit DiffGNN. Design and implementation of the synthetic graph generation process.

- **Chapter 6:** The experiments and data collection were conducted collaboratively, but all the data analysis, interpretation of the results, and conclusions are the result of my individual work.

Appendix A

Additional Charts

This appendix contains additional charts that were not included in the main body of the thesis due to space constraints.

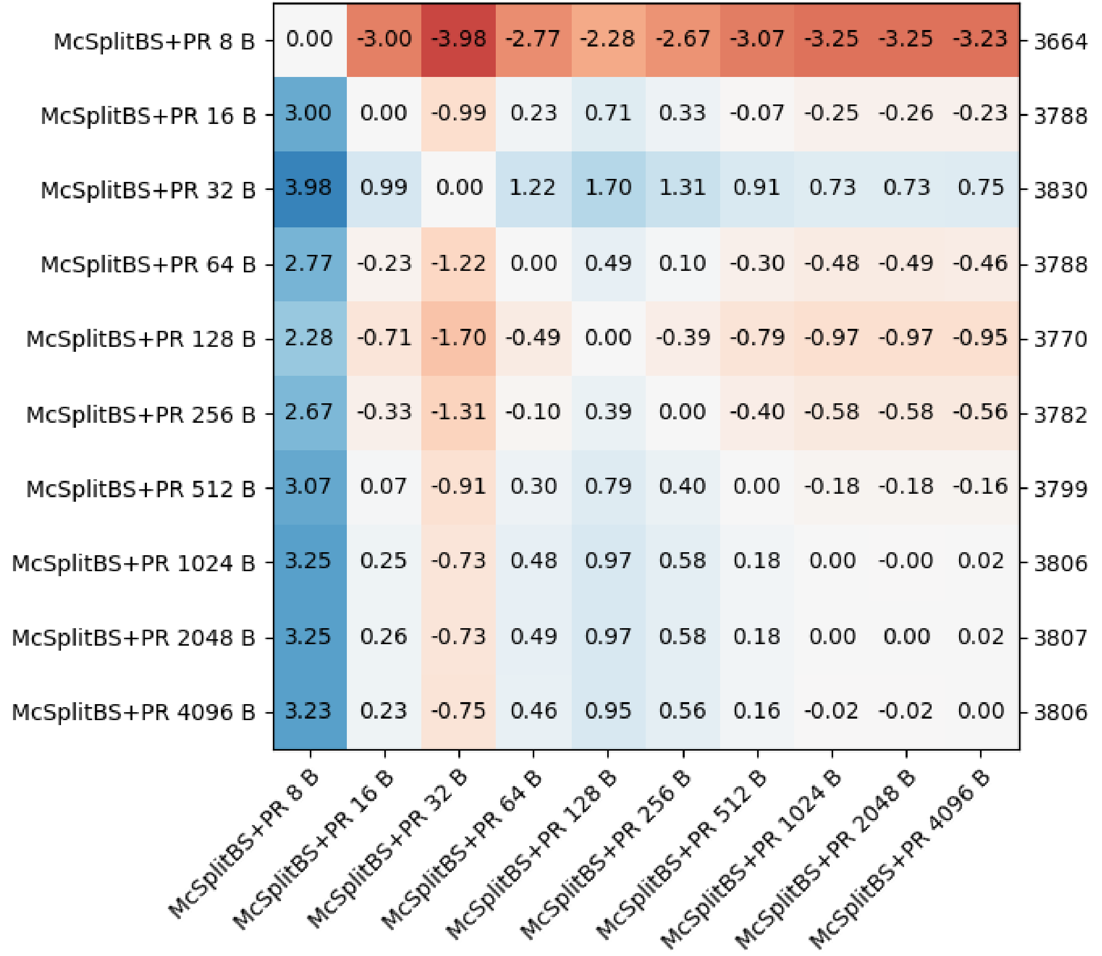


Figure A.1: Full difference heatmap of McSplitBS versions using PageRank, 32 threads, and different block sizes on LARGE-FINETUNING

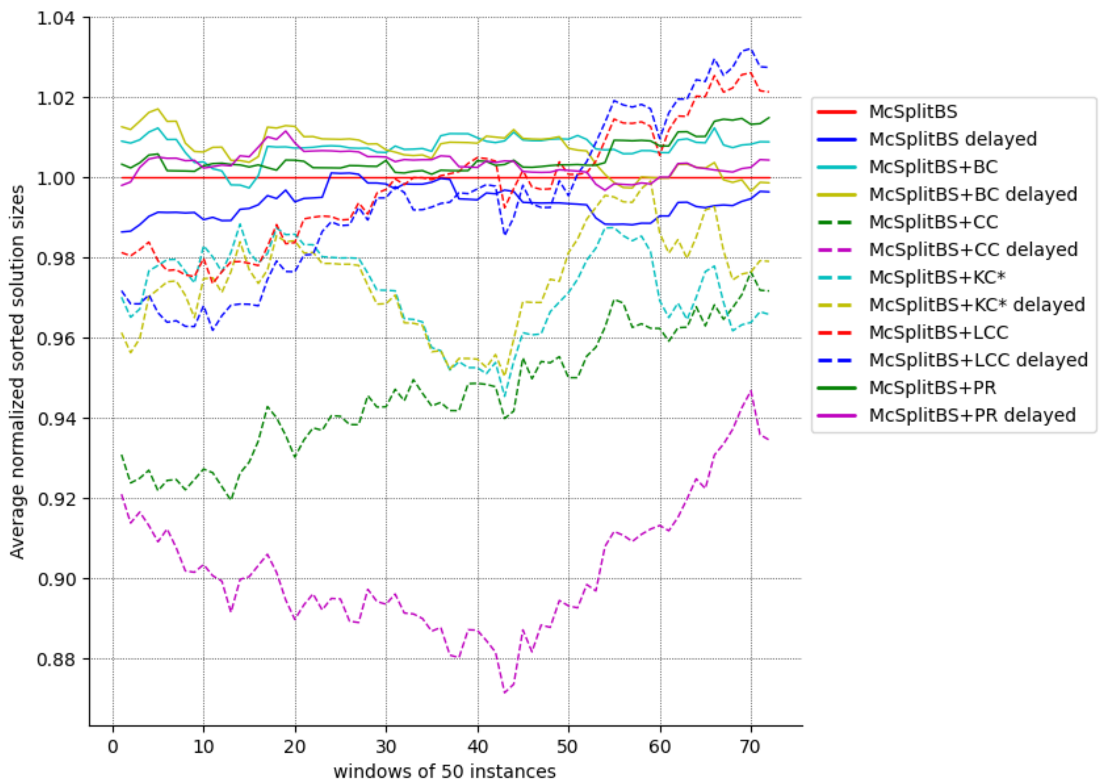


Figure A.2: Full comparison of McSplitBS versions using delayed sharing or not, with 32 threads, block size of 32, and different heuristics on Large-Finetuning

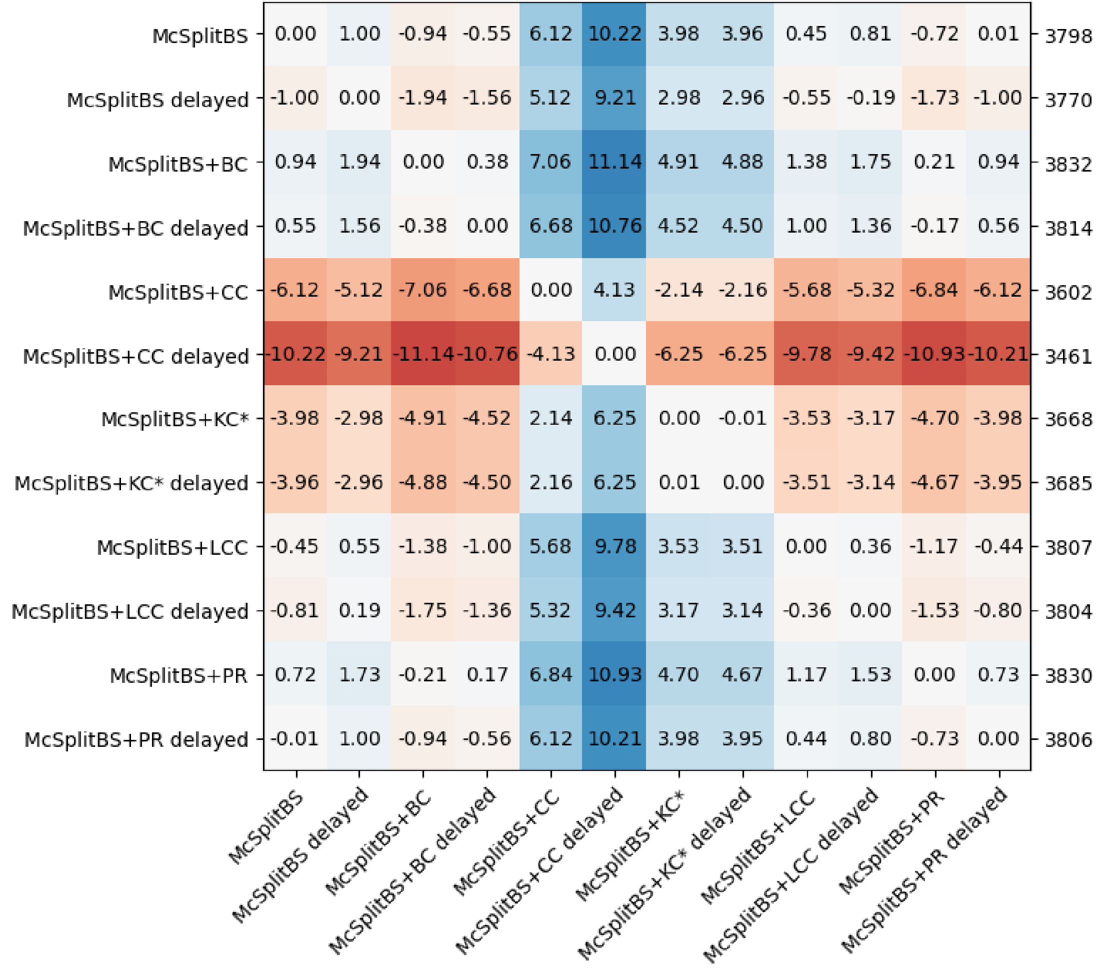


Figure A.3: Full difference heatmap of McSplitBS versions using delayed sharing or not, with 32 threads, block size of 32, and different heuristics on LARGE-FINETUNING

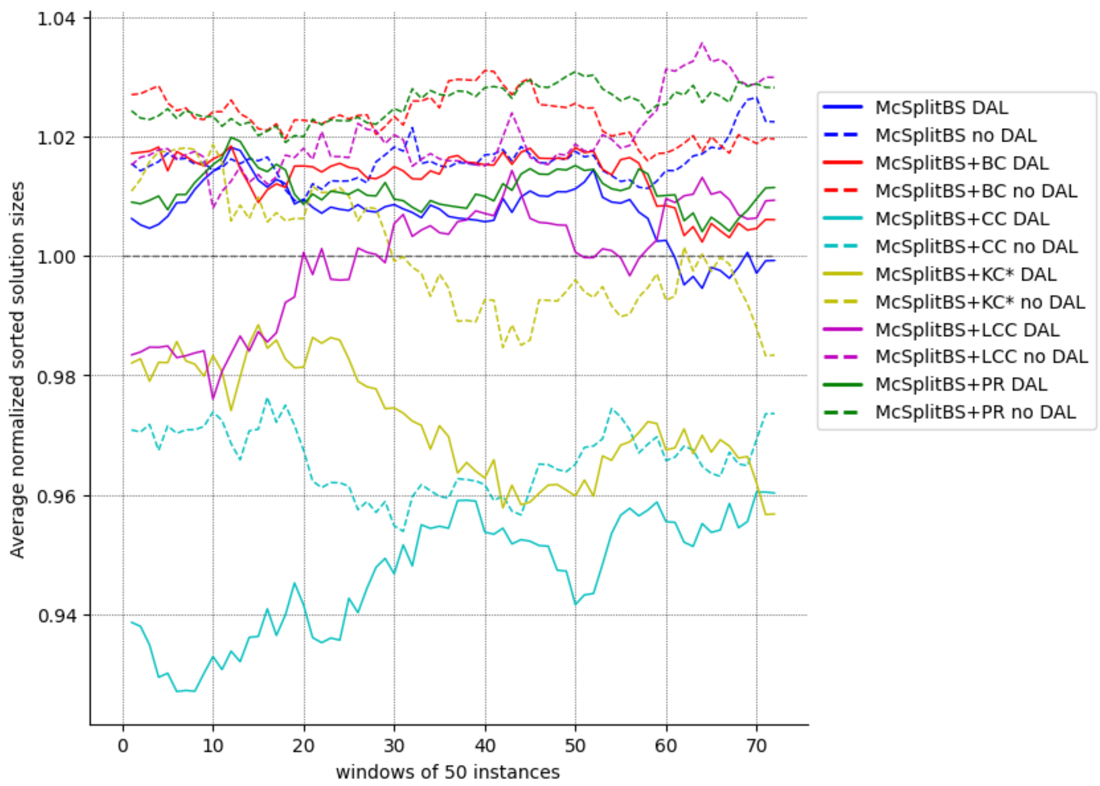


Figure A.4: Full comparison of McSplitBS versions using the DAL policy or not, with 32 threads, block size of 32, and different heuristics on Large-Finetuning

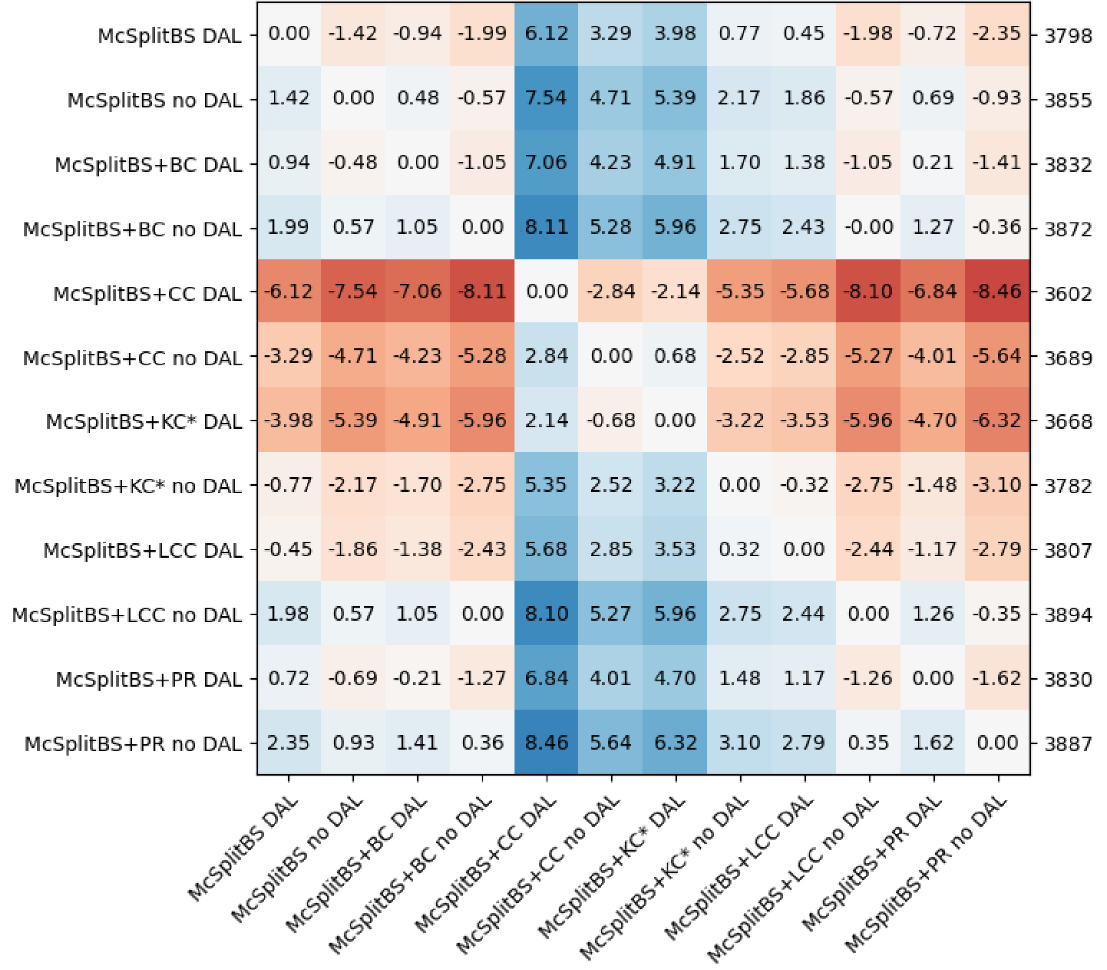


Figure A.5: Full difference heatmap of McSplitBS versions using the DAL policy or not, with 32 threads, block size of 32, and different heuristics on LARGE-FINETUNING

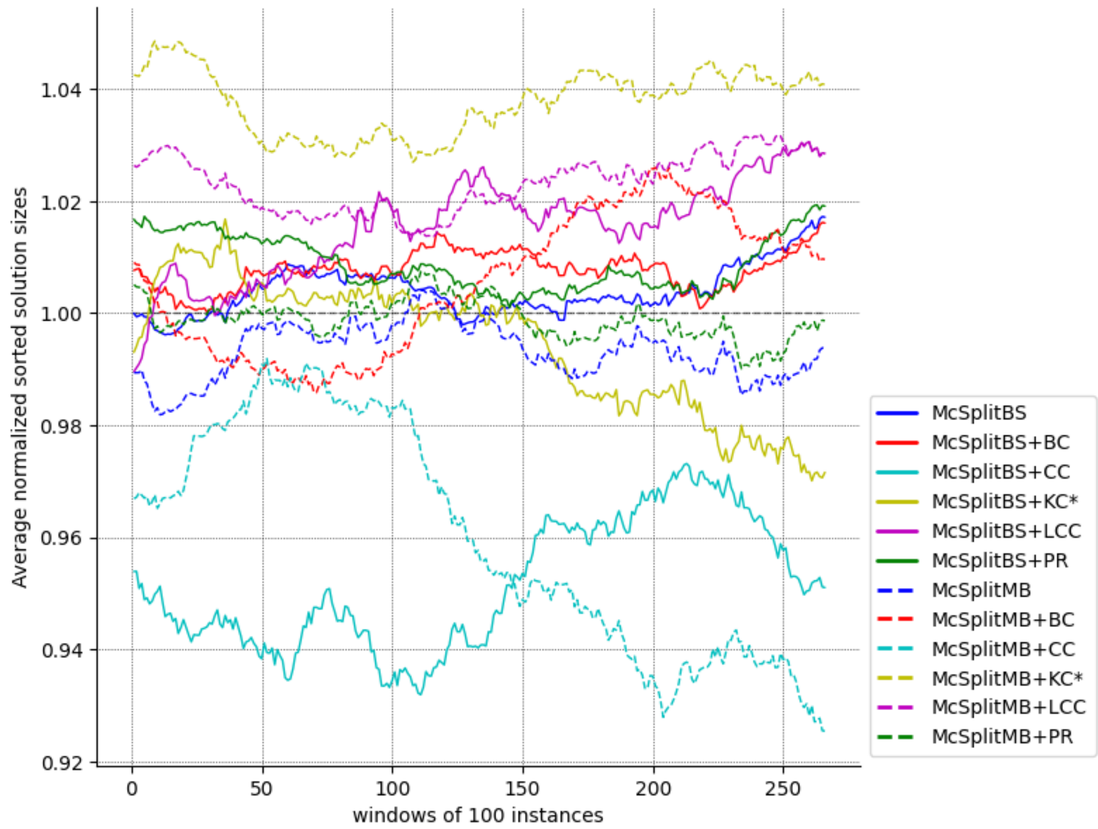


Figure A.6: Full comparison of McsplitMB and McSplitBS, with 32 threads, block size of 32, and different heuristics on Large

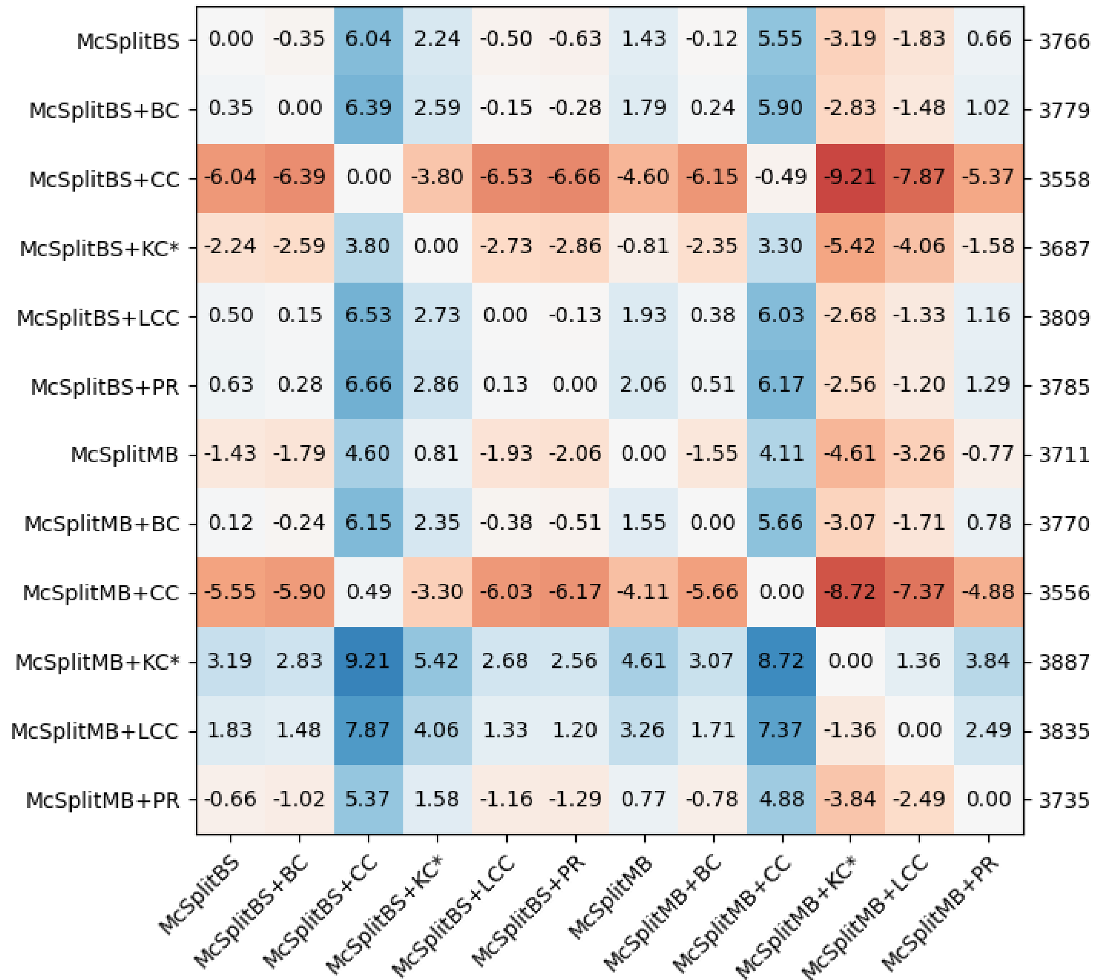


Figure A.7: Full difference heatmap of McsplitMB and McSplitBS, with 32 threads, block size of 32, and different heuristics on Large

Bibliography

[1] Leonhard Euler. «Solutio problematis ad geometriam situs pertinentis». In: *Commentarii academiae scientiarum Petropolitanae* 8 (1741), pp. 128–140 (cit. on p. 4).

[2] Bireswar Das, Murali Krishna Enduri, and I. Vinod Reddy. «Polynomial-time algorithm for isomorphism of graphs with clique-width at most three». In: *Theoretical Computer Science* 819 (2020). Computing and Combinatorics, pp. 9–23. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2017.09.013. URL: https://www.sciencedirect.com/science/article/pii/S0304397517306758 (cit. on p. 6).

[3] Charles J. Colbourn and Kellogg S. Booth. «Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs». In: *SIAM Journal on Computing* 10.1 (1981), pp. 203–225. DOI: 10.1137/0210015. eprint: https://doi.org/10.1137/0210015. URL: https://doi.org/10.1137/0210015 (cit. on p. 6).

[4] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974 (cit. on p. 6).

[5] Uwe Schöning. «Graph isomorphism is in the low hierarchy». In: *Journal of Computer and System Sciences* 37.3 (1988), pp. 312–323. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(88)90010-4. URL: https://www.sciencedirect.com/science/article/pii/0022000088900104 (cit. on p. 6).

[6] Stephen A. Cook. «The complexity of theorem-proving procedures». In: Association for Computing Machinery, May 1971, pp. 151–158. DOI: 10.1145/800157.805047 (cit. on p. 7).

[7] M R Garey and D S Johnson. «Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)». In: *Computers and Intractability* (1979), p. 340. ISSN: 0036-1445 (cit. on p. 8).

[8] Viggo Kann. «On the approximability of the maximum common subgraph problem». In: vol. 577 LNCS. 1992. DOI: 10.1007/3-540-55210-3_198 (cit. on p. 8).

[9] Peter J. Artymiuk, Ruth V. Spriggs, and Peter Willett. «Graph theoretic methods for the analysis of structural relationships in biological macromolecules». In: *Journal of the American Society for Information Science and Technology* 56 (5 2005). ISSN: 15322882. DOI: 10.1002/asi.20140 (cit. on p. 8).

[10] Hans Christian Ehrlich and Matthias Rarey. «Maximum common subgraph isomorphism algorithms and their applications in molecular science: A review». In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 1 (1 2011). ISSN: 17590876. DOI: 10.1002/wcms.5 (cit. on p. 8).

[11] John W. Raymond and Peter Willett. *Maximum common subgraph isomorphism algorithms for the matching of chemical structures.* 2002. DOI: 10.1023/A:1021271615909 (cit. on p. 8).

[12] Younghee Park, Douglas S. Reeves, and Mark Stamp. «Deriving common malware behavior through graph clustering». In: *Computers and Security* 39 (PART B 2013). ISSN: 01674048. DOI: 10.1016/j.cose.2013.09.006 (cit. on p. 8).

[13] Erick Nilsen Pereira de Souza, Daniela Barreiro Claro, and Rafael Glauber. «A similarity grammatical structures based method for improving open information systems». In: *Journal of Universal Computer Science* 24 (1 2018). ISSN: 09486968 (cit. on p. 8).

[14] Ciaran McCreesh, Patrick Prosser, and James Trimble. «A partitioning algorithm for maximum common subgraph problems». In: vol. 0. International Joint Conferences on Artificial Intelligence, 2017, pp. 712–719. ISBN: 9780999241103. DOI: 10.24963/ijcai.2017/99 (cit. on p. 9).

[15] Ruth Hoffmann, Ciaran McCreesh, and Craig Reilly. «Between Subgraph Isomorphism and Maximum Common Subgraph». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1 (Feb. 2017). DOI: 10.1609/aaai.v31i1.11137. URL: https://ojs.aaai.org/index.php/AAAI/article/view/11137 (cit. on p. 9).

[16] Matjaz Depolli, Sandor Szabo, and Bogdan Zavalnij. «An improved maximum common induced subgraph solver». In: *Match* 84 (1 2020). ISSN: 03406253 (cit. on p. 9).

[17] Yasuharu Okamoto. «Finding a Maximum Common Subgraph from Molecular Structural Formulas through the Maximum Clique Approach Combined with the Ising Model». In: *ACS Omega* 5 (22 2020). ISSN: 24701343. DOI: 10.1021/acsomega.0c00987 (cit. on p. 9).

[18] Runzhong Wang, Junchi Yan, and Xiaokang Yang. «Learning combinatorial embedding networks for deep graph matching». In: vol. 2019-October. 2019. DOI: 10.1109/ICCV.2019.00315 (cit. on p. 9).

[19] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. «Graph matching networks for learning the similarity of graph structured objects». In: vol. 2019-June. 2019 (cit. on p. 9).

[20] Information Technology. «Neural Maximum Common Subgraph Detection With Guided Subgraph Extraction». In: *System* (2009) (cit. on p. 9).

[21] Hongteng Xu, Dixin Luo, and Lawrence Carin. «Scalable gromov-wasserstein learning for graph partitioning and matching». In: vol. 32. 2019 (cit. on p. 9).

[22] Yunsheng Bai, Derek Xu, Alex Wang, Ken Gu, Xueqing Wu, Agustin Marinovic, Christopher Ro, Yizhou Sun, and Wei Wang. «Fast Detection of Maximum Common Subgraph via Deep Q-Learning». In: (Feb. 2020) (cit. on p. 9).

[23] James Trimble. «Partitioning algorithms for induced subgraph problems». PhD thesis. 2023 (cit. on pp. 13, 71).

[24] Martin L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. wiley, Jan. 2008, pp. 1–649. ISBN: 9780470316887. DOI: 10.1002/9780470316887 (cit. on p. 15).

[25] Matthew T. Regehr and Alex Ayoub. «An Elementary Proof that Q-learning Converges Almost Surely». In: *CoRR* abs/2108.02827 (2021). arXiv: 2108.02827. URL: https://arxiv.org/abs/2108.02827 (cit. on p. 15).

[26] Thomas G. Dietterich. «Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition». In: *Journal of Artificial Intelligence Research* 13 (2000). ISSN: 10769757. DOI: 10.1613/jair.639 (cit. on p. 16).

[27] Yanli Liu, Chu Min Li, Hua Jiang, and Kun He. «A learning based branch and bound for maximum common subgraph related problems». In: 2020. DOI: 10.1609/aaai.v34i03.5619 (cit. on p. 16).

[28] 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». In: 2022. DOI: 10.24963/ijcai.2022/265 (cit. on p. 17).

[29] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. *Graph Attention Networks*. 2018. arXiv: 1710.10903 [stat.ML] (cit. on p. 23).

[30] 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 p. 25).

[31] Lawrence Page and Sergey Brin. «The anatomy of a large-scale hypertextual Web search engine». In: *Computer Networks* 30 (1-7 1998). ISSN: 13891286. DOI: 10.1016/s0169-7552(98)00110-x (cit. on p. 31).

[32] Linton C. Freeman. «A Set of Measures of Centrality Based on Betweenness». In: *Sociometry* 40 (1 1977). ISSN: 00380431. DOI: 10.2307/3033543 (cit. on p. 33).

[33] Matthias Bentert, Alexander Dittman, Leon Kellerhals, André Nichterlein, and Rolf Niedermeier. «An adaptive version of brandes' algorithm for betweenness centrality». In: *Journal of Graph Algorithms and Applications* 24 (3 2020). ISSN: 15261719. DOI: 10.7155/jgaa.00543 (cit. on p. 33).

[34] Matthias Bentert, Alexander Dittman, Leon Kellerhals, André Nichterlein, and Rolf Niedermeier. «An adaptive version of brandes' algorithm for betweenness centrality». In: *Journal of Graph Algorithms and Applications* 24 (3 2020), pp. 483–522. ISSN: 15261719. DOI: 10.7155/jgaa.00543 (cit. on p. 35).

[35] Alex Bavelas. «Communication Patterns in Task-Oriented Groups». In: *Journal of the Acoustical Society of America* 22 (6 1950). ISSN: NA. DOI: 10.1121/1.1906679 (cit. on p. 35).

[36] Junzhou Zhao, Pinghui Wang, John C.S. Lui, Don Towsley, and Xiaohong Guan. «I/O-efficient calculation of H-group closeness centrality over disk-resident graphs». In: *Information Sciences* 400-401 (2017), pp. 105–128. ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2017.03.017. URL: https://www.sciencedirect.com/science/article/pii/S0020025517305960 (cit. on p. 35).

[37] Leo Katz. «A new status index derived from sociometric analysis». In: *Psychometrika* 18.1 (1953), pp. 39–43 (cit. on p. 35).

[38] Yunsheng Bai, Derek Xu, Yizhou Sun, and Wei Wang. «GLSearch: Maximum Common Subgraph Detection via Learning to Search». In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, 18–24 Jul 2021, pp. 588–598. URL: https://proceedings.mlr.press/v139/bai21e.html (cit. on pp. 54, 55).

[39] P. Foggia, C. Sansone, and Mario Vento. «A database of graphs for isomorphism and sub-graph isomorphism benchmarking». In: *Proc. of the 3rd IAPR TC-15 International Workshop on Graph-based Representations* (March 2001) (cit. on p. 66).

[40] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. June 2014 (cit. on p. 66).

[41] A. Calabrese, L. Cardone, S. Licata, M. Porro, and S. Quer. «A Web Scraping Algorithm to Improve the Computation of the Maximum Common Subgraph». In: vol. 0. 2023, pp. 197–206. ISBN: 978-989-758-665-1 (cit. on p. 80).