A Web Scraping Algorithm to Improve the Computation of the Maximum Common Subgraph

Andrea Calabrese[a], Lorenzo Cardone[b], Salvatore Licata, Marco Porro and Stefano Quer[c]
DAUIN Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy

Keywords: Graphs, Maximum Common Subgraph, McSplit, Heuristics, Software, Algorithms.

Abstract: The Maximum Common Subgraph, a generalization of subgraph isomorphism, is a well-known problem in the computer science area. Albeit being NP-complete, finding Maximum Common Subgraphs has countless practical applications, and researchers are continuously exploring scalable heuristic approaches. One of the state-of-the-art algorithms to solve this problem is a recursive branch-and-bound procedure called McSplit. The algorithm exploits an intelligent invariant to pair vertices with the same label and adopts an effective bound prediction to prune the search space. However, McSplit original version uses a simple heuristic to pair vertices and to build larger subgraphs. As a consequence, a few researchers have already focused on improving the sorting heuristics to converge faster. This paper concentrate on these aspects and presents a collection of heuristics to improve McSplit and its state-of-the-art variants. We present a sorting strategy based on the famous PageRank algorithm, and then we mix it with other approaches. We compare all the heuristics with the original McSplit procedure, and against each other. In particular, we distinguish the heuristics based on the node degree and novel ones based on the PageRank algorithm. Our experimental section shows that PageRank can improve both McSplit and its variants significantly regarding convergence speed and solution size.

1 INTRODUCTION

Graphs are flexible structures that allow us to model many elements of human knowledge through a mathematical abstraction. In particular, graphs can be very good representations of relationships between objects. Graphs find many applications in fields such as chemistry (Dalke and Hastings, 2013), social networks (Milgram, 1967), web searches (Brin and Page, 1998), security threat detection (Park and Reeves, 2011), modeling dependencies between different software components (Zimmermann and Nagappan, 2007), hardware testing and functional test programs (Angione et al., 2022).

In this paper, we are interested in improving the computation of the Maximum Common Subgraph (MCS) between two graphs. Even if the problem has been appearing in the scientific literature since the 70s (Bron and Kerbosch, 1973; Barrow and Burstall, 1976), one of the most efficient state-of-the-art algorithm for finding MCS is McSplit, introduced in 2017 by McCreesh et al. (McCreesh et al., 2017). McSplit is a branch-and-bound algorithm that recursively computes new solutions by pairing vertices selected from the two graphs. The core idea is to label all vertices based on the connection they have with already selected nodes. After that, the algorithm efficiently prunes the search tree taking into account those labels and a formula computing the upper bound for the size of the current solution. The approach is quite efficient in maintaining low memory profiles and pruning the search space. Unfortunately, it considers all possible vertex pairs, one vertex from the first and one from the second graph, and its performances strongly depend on the vertex sorting heuristic. The original version of McSplit statically sorts the vertices of both graphs based on their degree. This order is then maintained unaltered for the entire process, and it is the most impairing element of the procedure. Many vertices may have identical degrees, making it impossible to discriminate between them. Moreover, there is no way to prioritize a promising pair discovered during the execution of the algorithm. In our approach, we exploit the core of the original McSplit procedure, but we replace the static sorting heuristic with sharper ordering techniques.

[a] https://orcid.org/0000-0002-8854-8171
[b] https://orcid.org/0009-0008-7553-4839
[c] https://orcid.org/0000-0001-6835-8277

Calabrese, A., Cardone, L., Licata, S., Porro, M. and Quer, S.
A Web Scraping Algorithm to Improve the Computation of the Maximum Common Subgraph.
DOI: 10.5220/0012130800003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 197-206
ISBN: 978-989-758-665-1; ISSN: 2184-2833

McSplitRL (Liu et al., 2020), McSplitLL (Zhou et al., 2022), and McSplitDAL (Liu et al., 2022) already brought an improvement over the original sorting heuristic of McSplit. McSplitRL uses a Reinforcement Learning approach to refine the order of the vertex selection. McSplitLL, based on McSplitRL, outperforms its predecessor by using a technique called Long Short Memory which deals with nodes with specific characteristics. McSplitDAL builds upon McSplitLL, introducing a technique called Dynamic Action Learning, which improves the reward function of McSplitRL. However, these techniques use the original McSplit sorting heuristic as a tie-breaker when selecting vertices.

In this work, we present a new vertex selection heuristic that is able to improve the performances of McSplit, McSplitLL, and McSplitDAL. In particular, we propose to use PageRank (Brin and Page, 1998), the former algorithm behind the Google search engine, as a vertex selection heuristic, exploiting its capabilities to work on both directed and undirected graphs. We use PageRank both as a standalone or as a tie-breaking heuristic, using it to classify vertices and then combining it with other techniques such as McSplitLL or McSplitDAL.

In our experimental analysis, we compare our algorithm with McSplit and its variants. We tested 400 graph pairs, selecting the graphs from the largest publicly available graphs at (Foggia et al., 2001) and choosing at least one graph pair for each graph category. We set the timeout for each experiment to 60 seconds to quickly grab the convergence speed of each algorithm. Overall, we can improve McSplit, McSplitRL, McSplitLL, and McSplitDAL in up to 77% of the graph pairs considered. Moreover, we obtain an improvement in terms of the final size of the solution subgraph up to 7%.

The paper is organized as follows. In Section 2, we describe our notation and we define the problem. We also present a set of well-known approaches for solving it. In Section 3, we illustrate new heuristics to enhance the original McSplit algorithm and its latest variants. Section 4 describes our experimental results. Finally, Section 5 draws some conclusions and give some hints on possible future work.

2 BACKGROUND

This section introduces our graph notation and some basic concepts on subgraph isomorphism and the Maximum Common Subgraph problem. After that, we present McSplit and its more recent variants, which we consider state-of-the-art algorithms for solving the Maximum Common Subgraph problem.

2.1 Graphs

A graph is a pair of vertices (nodes) and edges (links). Links represent connections with nodes, making this structure well-suited for representing relationships between objects. In our notation, we use G and H to represent two graphs and $V(G)$ ($V(H)$) to represent the vertices belonging to G (H). Furthermore, we use $E(G)$ (and $E(H)$) to represent the set of all the pairs of vertices connected by an edge. We use $|G|$ or $|V(G)|$ to indicate the number of vertices belonging to G, referring to it as its *size*. In contrast, we refer to the number of edges of a graph as $|E(G)|$. Given $v_1, v_2 \in V(G)$, we denote $E(v_1, v_2)$ the edge that links v_1 to v_2.

Graphs can come in various flavors: Labeled or unlabeled, weighted or unweighted, directed or undirected. In labeled graphs, vertices have additional information described by the label; in many applications, the labels *classify* the vertices as sharing specific characteristics. In our notation, $L(v)$ is the label of the vertex v.

We say that the graph is *weighted* if edges present different weights associated with them. For example, a weight might represent the distance between two nodes. Unweighted graphs can be seen as weighted graphs with every weight equal to one.

We say that G is *undirected* if

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

In other words, if a link exists between v_1 and v_2, the opposite link must exist and have the same weight.

We say that H is a *subgraph* of G if

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

that is, the vertices and edges of H are a subset of the vertices and edges of G. A graph H is an *induced subgraph* of G if H is a subgraph of G and contains all the edges between its vertices of the original graph G.

Graph isomorphism is the problem of detecting if there is a bijection between two graphs G and H such that

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

that is, if two graphs have the same structure. Verifying whether two graphs are isomorphic is known to be NP (Schöning, 1988), even if the exact complexity inside that class is unknown.

A subgraph is a subset of a graph's vertices (or nodes) and edges (or links). The terms vertex and node will be used interchangeably in this paper.

The Maximum Common Subgraph (MCS) problem between graphs G and H, requires finding the most extensive graph simultaneously isomorphic to a subgraph of G and H. In particular, the Maximum Common Induced Subgraph (MCIS) focuses on finding the induced subgraph with all the vertices in common between two graphs. The problem is known to be NP-complete (Michael Garey, 1979).

In our case, we focus on undirected, unlabeled, and unweighted graphs, as they represent the worst case scenario for the Maximum Common Subgraph computation.

2.2 McSplit

McSplit (McCreesh et al., 2017) is a branch-and-bound recursive algorithm for finding the MCS between two graphs.

The authors define a *label class* as a set of vertex pairs (belonging to the first and the second graph) having the same connections toward the vertices belonging to the current solution. As McSplit uses labels to find possible couplings between vertices, the original algorithm also provides a way to create those labels based on the adjacency lists of the vertices.

```
1  $BEST \leftarrow \emptyset$
2
3  Function MCS($G, H, M$)
4      if $|M| > |BEST|$ then
5          $BEST \leftarrow M$
6      end
7      if $CalculateBound() < |BEST|$ then
8          return
9      end
10     $label\_class \leftarrow SelectLabelClass(G,H)$
11     $G' \leftarrow G$
12     while $G' \neq \emptyset$ do
13         $v \leftarrow SelectVertex(G, label\_class)$
14         $G' \leftarrow G' \setminus \{v\}$
15         forall
             $w \in getVertices(H, label\_class)$ do
16             $M' \leftarrow M \cup (v,w)$
17             $H' \leftarrow H \setminus \{w\}$
18             $G' \leftarrow UpdateLabels(G', v)$
19             $H' \leftarrow UpdateLabels(H', w)$
20             $mcs(G', H', M')$
21         end
22     end
23     $mcs(G', H, M)$
24     return
```

Algorithm 1: The simplified version of the original McSplit algorithm.

Algorithm 1 provides a simplified version of the McSplit algorithm. It takes as inputs the two graphs, G and H, as well as the current solution M. Label classes are used to guide the algorithm in finding the solution to the problem. The label class is a classification of each couple of vertices belonging to (G,H). First, the algorithm assigns the current solution to the best one (line 5), in case the current solution has a larger size (line 4). Notice that the best solution *BEST* is initially empty (line 1). Then, the algorithm calculates the upper bound B for the current path (line 7). If this upper bound is less than the size of the best solution, the current solution cannot be improved along the current path; thus, the algorithm backtracks (line 8). Otherwise, the algorithm keeps improving the current solution. The bound is computed as shown in Equation 1.

$$\begin{aligned} B \quad = \quad |M| + \textstyle\sum_{l \in L} \min(&|\{v \in G \backslash M : L(v) = l\}|, \\ &|\{w \in H \backslash M : L(w) = l\}|) \end{aligned} \tag{1}$$

When improving the current solution, McSplit tries to build a larger solution by virtually removing a couple of vertices with the same label from the respective graphs, updating the labels (lines 18-19) and trying to explore recursively all possibilities starting from the current solution (line 20). In each iteration of the algorithm, the selection of a vertex pair occurs in three distinct stages. Firstly, the most promising label class is identified, followed by selecting a vertex from the set of vertices belonging to that label class in the graph G (line 14). Subsequently, all vertices $w \in H$ of the chosen label class are gathered (line 15), and then individually selected one by one (line 17). Once $v \in G$ is selected, the current *mcs* instance uses recursion to explore all solutions that include v and all the nodes of the received partial solution M, therefore at line 23 an additional recursive call is introduced to explore all the other solutions that include M but exclude v. Ultimately, as every vertex couple has been explored (line 12), the procedure returns the best solution.

To explore all possible vertex pairs, McSplit uses two different heuristics. The first one is used to select the next label class. The second one is adopted to choose the next vertex to add to the final graph. The former (line 10) chooses the label class with the smallest maximum size between G and H, i.e., $max(|G|,|H|)$. The latter, instead, prioritizes vertices in G with the most significant degree, where the degree is the number of links (inward and outward) of the vertex. In particular, for selecting the next vertex (line 13), McSplit heuristically considers the degree of the vertex, choosing each time the vertex with the most considerable degree and removing it from the graph. We will refer to this approach as the *Node De-*

gree, or simply the *Degree* heuristic.

2.3 McSplit Variants

Many notable variants of McSplit have been developed to improve over the original algorithm. This section briefly describes some of the most noticeable and recent ones.

2.3.1 McSplitSD

McSplit works asymmetrically on the two graphs since it selects a vertex from G and then searches for a matching vertex in H. This approach may unbalance the algorithm, making it perform better or worse, depending on the characteristics of the first graph. Among other strategies, Trimble (Trimble, 2023) proposes McSplitSD, which sets as the first graph the denser one of the pair. The density K of a graph is evaluated through Equation 2, using the number of edges and vertices of the two graphs to express the *density extremeness*:

$$K(G) = \frac{|E(G)|}{|V(G)| \cdot (|V(G)|-1)} \tag{2}$$

The two graphs G and H are swapped when the inequality

$$\left|\frac{1}{2} - K(G)\right| > \left|\frac{1}{2} - K(H)\right|$$

is true.

2.3.2 McSplitRL

Liu et al. (Liu et al., 2020) proposes McSplitRL, a novel approach that extends the standard McSplit using Reinforcement Learning. This approach keeps two vectors, one for the vertices of G and the other for the vertices of H, which contain the rewards of each node. Therefore, the node selection heuristic is based on finding the node with the highest reward. The authors devised a scoring system for a given action using Equation 3:

$$\begin{aligned} R(v,w) = & \textstyle\sum_{(V_l,V_r)\in E_v} \min(|V_l|,|V_r|) - \\ & \textstyle\sum_{(V'_l,V'_r)\in E_{v'}} \min(|V'_l|,|V'_r|) \end{aligned} \tag{3}$$

Given a set of label classes of the initial graphs at a given point of the search, E_v, and the subsequent set of label classes, E'_v, generated by including a new couple of vertices to the current solution, Equation 3 calculates the reduction of the size of the label classes. The size of a label class is considered as the minimum of $|V_l|$ and $|V_r|$, which are the number of vertices belonging to the label class respectively from the first or the second graph. Thus, this method can be seen as a bound reduction and tends to prefer nodes whose resulting branching cause a higher reduction of the bound, thus cutting as many branches as possible in subsequent steps of the algorithm.

2.3.3 McSplitLL

Zhou et al. (Zhou et al., 2022), starting from McSplitRL, build a more sophisticated version of the tool called McSplitLL. Their solution introduces a new heuristic called Long Short Memory (LSM) and a method to be used in a specific situation called Leaf Vertex Union Match (LUM). The new heuristic uses Equation 3 but stores the rewards in a vector for nodes of G and a matrix for the nodes of H, allowing to reward each possible node pair separately $(v,w) \in (G,H)$.

However, since rewards may become huge, an asymmetric decay is used, following a long-short-term approach, which halves both G and H rewards when their respective thresholds are exceeded. Rewards for single nodes v decay faster than the rewards for pair of nodes (v,w); thus, node pairs have a smaller threshold.

Moreover, the LUM heuristic introduces a more optimized strategy to handle leaf nodes. A node is considered a leaf if it is adjacent to only one vertex of a given graph, and it has been proved it can always be added to the current subgraph if its only neighbor is part of it as well. Thus, whenever a leaf from the left graph and a leaf from the right graph is found, the pair formed by these two nodes is added to the current solution.

2.3.4 McSplitDAL

Liu et al. introduced McSplitDAL (Liu et al., 2022). This algorithm is the most recent version of McSplit, and it is built upon McSplitRL and McSplitLL. This algorithm mainly introduces two new ideas. A new value function called Domain Action Learning (DAL) and a hybrid learning policy for choosing the next vertex to match. The DAL value function aims to take into account, when branching, not only the reduction of the upper bound but also the simplification of the problem occurring after the branch. This feature can be implemented by adding an additional term to the reward defined in Equation 3, granting a higher reward to the vertices whose generated partitions have a higher cardinality, when these vertices are added to the solution:

$$\begin{aligned} R(v,w) = & \textstyle\sum_{(V_l,V_r)\in E_v} \min(|V_l|,|V_r|) - \\ & \textstyle\sum_{(V'_l,V'_r)\in E_{v'}} \min(|V'_l|,|V'_r|) + \\ & |E_{v'}| \end{aligned} \tag{4}$$

Moreover, the hybrid branching policy of this approach has the primary goal of overcoming a possible “Matthew effect”, which causes the algorithm to continue branching on a subset of nodes with very high rewards getting trapped in a local optimum. The authors believe this can be overcome by switching from the RL to the DAL policy (and vice versa) after a fixed number of iterations without improvement, allowing to dynamically change the strategy for selecting nodes.

For brevity, in this paper, we use the term *McSplitX* to generically identify the original McSplit or one of its variants, i.e., McSplitLL, or McSplitDAL.

2.4 Other Approaches

Many algorithms have been presented to solve the MCS problem, using strategies that differ from the original McSplit. Among those, we would like to mention the following. Levi (Levi, 1973) casts the MCS problem onto the Maximum Common Clique problem. McCreesh et al. (McCreesh et al., 2016) and Vismara et al. (Vismara and Valery, 2008) follow the previous approach while exploiting constraint programming to solve the problem. Other approaches take a step back, adopting parallel computation capabilities of General-Purpose computing on Graphics Processing Unit (GPGPU) (Quer et al., 2020), to enhance McSplit on modern devices. A set of heuristics to tackle the MCS problem with more than two graphs has been developed by Cardone et al. (Cardone and Quer, 2023). However, the most promising heuristics work by analyzing graphs in couples and later merging the results, thus still motivating the research on MCS techniques working on pairs of graphs.

3 OUR APPROACH

The main target of this work is to improve the vertex selection heuristic. In particular, we are interested in heuristics that can classify the vertices of the two graphs. From our perspective, a good heuristic should follow the guidelines presented by Marti et al. (Martí and Reinelt, 2022):

- The solution should be nearly optimal.
- The heuristic should require low computational effort.

In our heuristics, we also aim to generate classifications as diverse as possible for ranking the vertices. Moreover, we would like heuristics to classify a vertex with a single number instead of representing it as a vector. Although vectors have already been used in MCS solutions, due to the nature of the problem, using a mathematical vector incurs possible downfalls. More specifically, vectors may require more computational power to retrieve a classification than using single integers and the results may depend on the lexicographical order of the vertices. With these considerations in mind, we focus on a classification of vertices based on single numbers. In particular, we developed different heuristics for classifying vertices:

- A heuristic considering the PageRank of each vertex.
- A heuristic using both PageRank and McSplit-DAL.
- A heuristic using both PageRank and McSplitLL.

Please notice that both DAL and LL heuristics are computed dynamically, whereas the PageRank approach is applied only once at the beginning of the procedure.

3.1 The PageRank Algorithm

PageRank (Brin and Page, 1998) is an algorithm developed by Google that, given a network of web pages, generates the probability of reaching a page through a finite sequence of random clicks. PageRank was the algorithm used by Google to sort the results of its web engine searches. However, it is not used anymore, as its patent expired in 2019.

PageRank is usually implemented on a generic graph, so to account for different web pages, it considers directed and unweighted graphs. A link from one web page takes the user to another web page, but the way back is not guaranteed. However, we can also use it on undirected graphs, as we can think of them as directed graphs with both forward and backward edges between each node pair.

Algorithm 2 implements our PageRank algorithm, and it is strongly inspired by a public version[1]. In Algorithm 2, we use the notation $adj(G)$ to refer to the indices of the adjacency matrix of graph G.

The Damping Factor (DF), initialized in line 1, represented a person's probability of stopping clicking random links. We decided to follow Brin et al. (Brin and Page, 1998) recommendation for the value of the DF, and we set its value at 0.85. In line 2, we set the acceptable error ε at an arbitrary value. Experimentally, we discover that the smaller the epsilon (i.e., the more we increase the precision of the procedure), the better the results, as the rankings tend to be more diverse. However, as the original algorithm accepts integers numbers, we also want to be able to map integers to ranks; thus, we chose for ε a precise enough number that would surely not overflow any 32-bit integer.

PageRank can be described as a Markov chain. Thus, we build a stochastic matrix representing the graph in line 17, based on the previously computed

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

links going out from each node in line 16. Computing the outgoing links is trivial and is not shown in the algorithm. On the contrary, the computation of the stochastic matrix is represented in function `StochasticGraph`, from line 4 to line 13. Assuming that each node has a unitary amount of information flowing outwards to the neighbors, the matrix identifies how much of that information is flowing through each of the adjacent edges. In line 18 we transpose the stochastic matrix, and outgoing links are replaced with incoming links and vice versa. PageRank ranks nodes based on their incoming links; thus, the inversion is necessary for the generality of the algorithm. For undirected graphs, this might represent an unnecessary step; however, as McSplit works on directed and undirected graphs, this must be true also for its intermediate stages. On line 20, we pre-allocate the results of the previous iteration and set them to zero.

In line 22 we calculate the ratio between the incoming or outgoing links and the size of the graph. The core section of the evaluation is included from line 25 to line 38. First, we zero the results for the current iteration. Then, we compute the current rank by adjusting the previous results, approximating at each iteration the clicking probability, and discounting them by the *DF*. On line 35, we update the error on the measurement, and on line 37 we update the result vector *p*. The algorithm terminates when ($error < \varepsilon$) in line 25; this condition is triggered when the rankings converge, reaching a stable configuration.

As we consider it trivial, we do not show the float to integer conversion in Algorithm 2.

```
1  $DF \leftarrow 0.85$
2  $\varepsilon \leftarrow 0.00001$
3
4  **Function** StochasticGraph($G, out\_links$)
5      $G_s \leftarrow [0.0] * |G|$
6      **forall** $x, y \in adj(G)$ **do**
7          **if** $out\_link[x] = 0$ **then**
8              $G_s[x,y] \leftarrow 1.0/|G|$
9          **else**
10             $G_s[x,y] \leftarrow G[x,y]/out\_link[x]$
11         **end**
12     **end**
13     return $G_s$
14
15 **Function** PageRank($G$)
16     $out\_links \leftarrow OutLinksForEachNode(G)$
17     $G_s \leftarrow StochasticGraph(G, out\_links)$
18     $G_t \leftarrow TransposeMatrix(G_s)$
19     $result \leftarrow \emptyset * |G|$
20     $p \leftarrow \emptyset$
21     **forall** $x, y \in adj(G_t)$ **do**
22         $push(G_t[x,y]/|G|)$
23     **end**
24     $error \leftarrow 1.0$
25     **while** $error > \varepsilon$ **do**
26         $result \leftarrow \emptyset * |G|$
27         **forall** $x, y \in adj(G_t)$ **do**
28             $result[x] \leftarrow$
                   $result[x] + G_t[x,y] * p[y]$
29         **end**
30         **forall** $rank \in result$ **do**
31             $rank \leftarrow rank * DF + \frac{1.0 - DF}{|G|}$
32         **end**
33         $error \leftarrow 0.0$
34         **forall** $rank, prev \in zip(results, p)$ **do**
35             $error \leftarrow error + abs(rank - prev)$
36         **end**
37         $p = result$
38     **end**
39     return $result$
```

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

3.2 McSplitX+PR

Within the framework introduced in Section 3.1, we exploit the ideas introduced by McSplitLL and McSplitDAL, enhanced by the integration of the PageRank heuristic. The union of these techniques produced two new versions of the McSplit algorithm, specifically referred to as McSplitLL+PR and McSplitDAL+PR.

Whilst the original McSplit idea was centered around the node degree heuristic, the subsequent variants were mainly based on McSplitRL, which used reinforcement learning as a vertex selection heuristic. However, whenever a tie is encountered, the heuristic falls back to the node degree for choosing a vertex.

We propose using PageRank as a standalone or tie-breaking heuristic, substituting it for the node degree. This approach is summarized by Algorithm 3. First, we apply the PageRank to classify the vertices of graphs *G* and *H* (in lines 2 and 3, respectively). Then, we sort the vertices following their ranks obtained by the previous classification (lines 4 and 5). Finally, we apply our McSplitLL or McSplitDAL (i.e., *McSpliX*, generically speaking) on the sorted vertices (line 6). This method leverages the Reinforcement Learning, to choose vertices dynamically along the search, and guarantees the use of the PageRank scores as a tie-breaker, particularly at the beginning of the algorithm, when the rewards are initialized

```
1 Function McSplitX+PR(G, H)
2     $G_{ranks} \leftarrow PageRank(G)$
3     $H_{ranks} \leftarrow PageRank(H)$
4     $G_{sorted} \leftarrow SortGraph(G, G_{ranks})$
5     $H_{sorted} \leftarrow SortGraph(H, H_{ranks})$
6     $McSplitX(G_{sorted}, H_{sorted})$
7     return
```

Algorithm 3: The proposed McSplitX+PR algorithm optimizing a McSplitX implementation recalled in line 6.

to zero.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We ran our tests on a workstation with an Intel Core i9-10900KF CPU and 64 GBytes of DDR4 RAM.

All our algorithms are written in C++, and we compiled it with GCC version 9.4. For McSplit and McSplitLL, we use the original versions obtained from the WEB and adapted for being used with our new heuristic. For McSplitDAL, we wrote an implementation that follows the ideas indicated by the authors (Liu et al., 2022) as we were unable to find an official version publicly available. In addition, since it has been proven to be beneficial, we borrow the graph swap idea from McSplitSD (Trimble, 2023), and include it in all the variants of McSplit. Our core implementation adopts the C++ parallel version of McSplit. Unfortunately, not all versions may run in multi-threading mode. Thus, as we are interested in comparing our results with the ones gathered with the previous variants of McSplit, we present all results running all parallel versions with a single thread.

All algorithms were tested on a publicly available dataset (Foggia et al., 2001). We focused on the most extensive graphs, the ones with 100 nodes. Given the size of the set, we chose at least one experiments for each graph category, finally selecting 400 graph pairs.

Our tests are designed to evaluate the most practical aspect of all algorithms; thus, we evaluate their ability to find suitable solutions in a limited amount of time, instead of finding the optimal solution with an unlimited timeout. For each graph pair, we then record the size of the most significant solution found. We compare the different methodologies in terms of their capacity to find the largest solution in the slotted time.

We fixed the timeout to 60 seconds for each experiment. This timeout has been selected because experimentally McSplit often finds an effective solution along the first recursion path and it improves it only sporadically. Figure 1 plots the typical growth of the solution size with respect to the number of recursions. We can see that at the beginning (within a few thousand of recursions, usually performed in less than one second in our setup) the solution size increases very rapidly. Unfortunately, after the first few seconds, the solution grows slowly as most of the time is spent searching the enormous solution space. In orange, we highlighted the solution size at the end of the recursion process. Please, notice that the number of recursions is reported on the x-axis on a logarithmic scale.

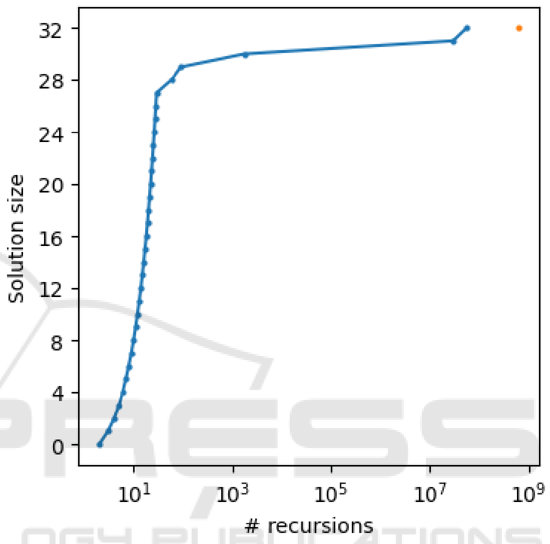

Figure 1: Typical behavior of the effectiveness of the original implementation of McSplit. The size of the solution often increases rapidly in the first part of the process; then, the procedure is captured by local minima which slow down the convergence process and force the algorithm to visit enormous state spaces that do not improve the solution size. In orange, we can see the solution size at the end of the execution.

4.2 Experimental Evaluation

Figure 2 reports the number of graph pairs on which each method finds the largest MCS out of the 400 graph experiments run. When an MCS with the same size is returned by more than one heuristic (i.e., we have a ex aequo) that pair is assigned to all the methods returning that result.

It is straightforward to see that our PR heuristic, only applied to McSplit, McSplitLL, and McSplitDAL, easily outperforms the original strategies. Moreover, the fastest strategy, i.e., McSplitDAL+PR, finds the most significant solution in almost 300 cases out of 400.

Table 1, using no tie-breaker, shows the percent-

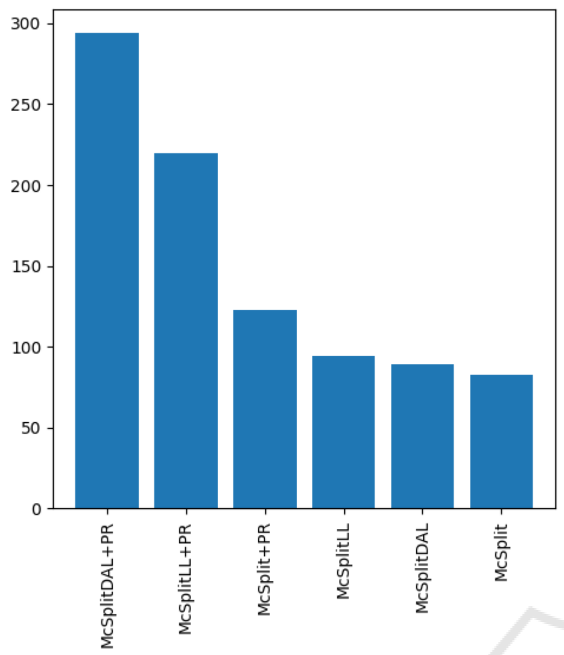

Figure 2: The histogram plots the number of times each heuristic finds the MCS (i.e., the largest maximum common subgraph) on the 400 experiments. When a graph with the same size is returned by more than one method, each strategy is reported as a winner.

age of victories of all PR-improved strategies with respect to each original method.

Table 1: Percentage of instances improved by the PR methods (columns) over the original methods (rows), without breaking ties.

Heuristics	**McSplit + PR** [%]	**McSplitLL + PR** [%]	**McSplitDAL + PR** [%]
McSplit	64	72	77
McSplitLL	60	69	76
McSplitDAL	63	72	77

Figure 2 and Table 1 focus on the number of experiments on which PageRank could return larger solutions than the original algorithms. Overall, they show that PR methods provide larger solutions for most of cases. However, we can also compare the size of the different solutions to understand the average improvements. To highlight the size of the results, we collected the size of the best solution found by each algorithm for every graph pair. To account for the natural variation in solution sizes between a wide range of instances of different complexity, we normalized all results with respect to the size of the subgraph found by the original McSplit algorithm.

In Figure 3, we show the average performance of our normalized heuristics. Due to the significant differences in solution sizes across instances, we plot a circular rolling average with a window size of 50 to better present the outcomes of our experiments. This strategy implies that each point on the plot represents the average normalized performance over a window of 50 consecutive tests. Due to the normalization, the original McSplit always returns solutions of size one, whereas all other methods almost always return more extensive solutions. Notably, PageRank demonstrates a distinct advantage over the degree heuristic. Moreover, McSplitDAL+PR and McSplitLL+PR methods consistently outperform their McSplitX counterparts in any batch of 50 instances and when they fall behind, they do not fall behind by a large amount.

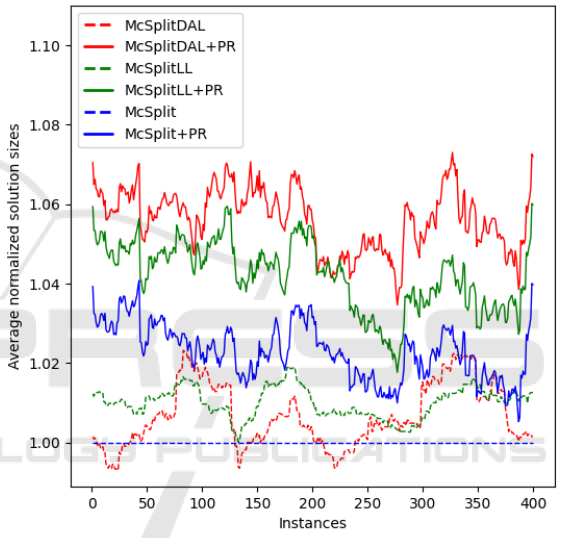

Figure 3: A circular rolling average (with a window width of 50 consecutive tests) of the sizes of the solutions obtained by the McSplitX and McSpliX+PR algorithms on each instance. All values are normalized with respect to the results obtained by the original McSplit.

The heat-map in Figure 4 shows the relative performance across all combinations of the algorithms. For each method on the vertical axis, the results are individually normalized with respect to the results of the algorithm on the horizontal axis; then, all the normalized values are averaged together.

From the map, we learn that McSplitDAL+PR exhibits an average improvement of 6% over McSplitDAL, McSplitLL+PR yields solutions that are 4% larger compared to McSplitLL, and McSplit+PR produces solutions 3% larger than McSplit. These results suggest that PageRank is an effective standalone heuristic, providing even more significant benefits when used as a tie-breaker on top of more complex Reinforcement Learning rewards.

It has to be noticed that in our testing, the McSplitDAL policy is not always better than the McSplitLL, unlike what was observed by Liu et al. (Liu et al., 2022). This result is likely due to our different evaluation methodologies. However, McSplitDAL+PR benefits from the PageRank heuristic, convincingly outperforming both McSplitLL and McSplitLL+PR by 6% and 2%, respectively.

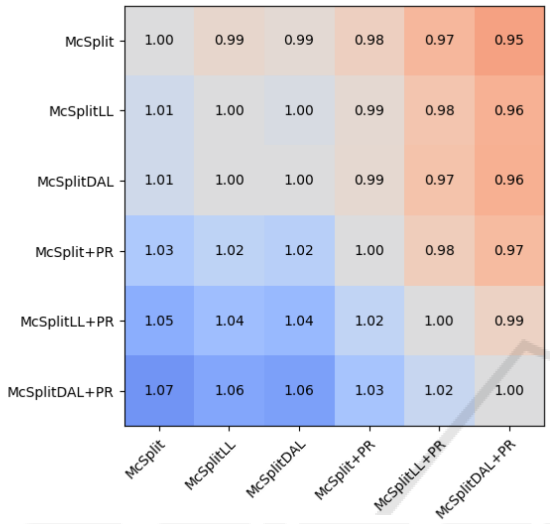

Figure 4: The relative performance of the McSplitX and McSplitX+PR methods. For each row, we report the average improvement relative to the respective column. Darker blue colors highlight the size improvements.

In Figure 5 we present a comprehensive comparison of the solution sizes achieved by each McSplitX+PR method and its corresponding McSplitX counterpart. For each instance, a dot is reported to show the size of the solutions found by the two algorithms. By removing the need for the rolling average, this scatter plot offers a better view of the results of the individual instances. Notably, the PageRank heuristic is the winner in most cases, particularly in the McSplitDAL+PR variant. Upon careful examination, it becomes evident that the average performance of the McSplitX methods is influenced by a few outlier instances that exhibit exceptional results. However, in contrast, McSplitX+PR consistently demonstrates improved performance across the entire range of instances.

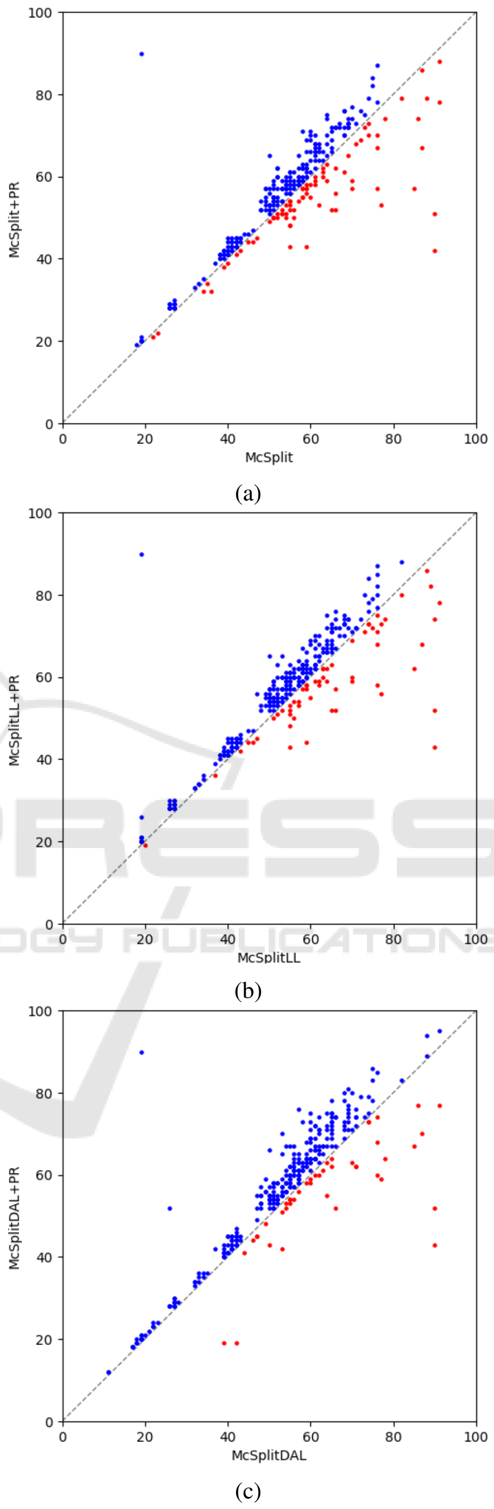

Figure 5: The dispersion of the points above the main diagonal shows that McSplitX+PR finds more extensive solutions in the vast majority of the cases.

5 CONCLUSIONS AND FUTURE WORKS

In this paper, we focus on solving the Maximum Common Induced Subgraph problem. Starting from a state-of-the-art algorithm called McSplit, and its recent variants (namely McSplitLL, McSplitRL, and

McSplitDAL). we propose a family of Branch-and-Bound algorithms called McSplitX+PR.

The original McSplit algorithm uses a node degree heuristic to select the vertices of the graphs during the recursive search. McSplitRL and its derivatives use rewards obtained through Reinforcement Learning, but still enforce the node degree to break ties. We propose the McSplitX+PR algorithm family, namely McSplit+PR, McSplitLL+PR, and McSplitDAL+PR, to replace the original node degree heuristic with the ranking produced by the PageRank algorithm. PageRank, famously known as the former algorithm behind the Google search engine, generates more effective node orderings compared to the degree of vertices, as it prioritizes nodes that are easier to reach across multiple hops rather than just in the local neighborhood, effectively differentiating them over more categories than the original heuristic.

Using publicly available graph pairs, we conducted experiments on both the McSplitX+PR and McSplitX families. We mainly focus on finding the best solution within a limited time to simulate real-world scenarios. Our results indicate that all McSplitX+PR algorithms consistently outperform their McSplitX counterparts, with McSplitDAL+PR yielding the most effective solutions than the other strategies.

Among the possible future works, we would like to mention the necessity of studying the multi-threaded versions of the above tools. In this work, this analysis has been limited by the fact that not all the considered tools were initially implemented with multi-threading capabilities. Consequently, one of our targets is to improve the above heuristics obtaining uniform scalability on multi-core architectures.

REFERENCES

Angione, F., Bernardi, P., Calabrese, A., Cardone, L., Niccoletti, A., Piumatti, D., Quer, S., Appello, D., Tancorre, V., and Ugioli, R. (2022). An innovative strategy to quickly grade functional test programs. In *2022 IEEE International Test Conference (ITC)*, pages 355–364.

Barrow, H. G. and Burstall, R. M. (1976). Subgraph Isomorphism, Matching Relational Structures and Maximal Cliques. *Inf. Process. Lett.*, 4(4):83–84.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1):107–117. Proceedings of the Seventh International World Wide Web Conference.

Bron, C. and Kerbosch, J. (1973). Finding All Cliques of an Undirected Graph (algorithm 457). *Commun. ACM*, 16(9):575–576.

Cardone, L. and Quer, S. (2023). The multi-maximum and quasi-maximum common subgraph problem. *Computation*, 11(4).

Dalke, A. and Hastings, J. (2013). Fmcs: a novel algorithm for the multiple mcs problem. *Journal of cheminformatics*, 5(Suppl 1):O6.

Foggia, P., Sansone, C., and Vento, M. (2001). A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In -, page 176–187.

Levi, G. (1973). A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO*, 9(4):341–352.

Liu, Y., Li, C.-M., Jiang, H., and He, K. (2020). A learning based branch and bound for maximum common subgraph related problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(03):2392–2399.

Liu, Y., Zhao, J., Li, C.-M., Jiang, H., and He, K. (2022). Hybrid learning with new value function for the maximum common subgraph problem.

Martí, R. and Reinelt, G. (2022). *Heuristic Methods*, pages 27–57. Springer Berlin Heidelberg, Berlin, Heidelberg.

McCreesh, C., Ndiaye, S. N., Prosser, P., and Solnon, C. (2016). Clique and constraint models for maximum common (connected) subgraph problems. In Rueher, M., editor, *Principles and Practice of Constraint Programming*, pages 350–368, Cham. Springer International Publishing.

McCreesh, C., Prosser, P., and Trimble, J. (2017). A partitioning algorithm for maximum common subgraph problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 712–719.

Michael Garey, D. S. J. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, United States.

Milgram, S. (1967). The small world problem. *Psychology today*, 2(1):60–67.

Park, Y. and Reeves, D. (2011). Deriving common malware behavior through graph clustering. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 497–502.

Quer, S., Marcelli, A., and Squillero, G. (2020). The maximum common subgraph problem: A parallel and multi-engine approach. *Computation*, 8(2).

Schöning, U. (1988). Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312–323.

Trimble, J. (2023). *Partitioning algorithms for induced subgraph problems*. PhD thesis, University of Glasgow.

Vismara, P. and Valery, B. (2008). Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In *Modelling, Computation and Optimization in Information Systems and Management Sciences: Second International Conference MCO 2008, Metz, France-Luxembourg, September 8-10, 2008. Proceedings*, pages 358–368. Springer.

Zhou, J., He, K., Zheng, J., Li, C.-M., and Liu, Y. (2022). A strengthened branch and bound algorithm for the maximum common (connected) subgraph problem.

Zimmermann, T. and Nagappan, N. (2007). Predicting subsystem failures using dependency graph complexities. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pages 227–236.