Improving state-of-the-art vertex sorting algorithms to compute the maximum common induced subgraph

Journal of Heuristics (2026) 32:1
https://doi.org/10.1007/s10732-025-09575-0

Improving state-of-the-art vertex sorting algorithms to compute the maximum common induced subgraph

Andrea Calabrese[1] · Lorenzo Cardone[1] · Salvatore Licata[1] · Marco Porro[1] · Stefano Quer[1]

Received: 17 December 2024 / Revised: 24 October 2025 / Accepted: 12 November 2025

Abstract

The Maximum Common Induced Subgraph problem is a longstanding challenge in graph theory and combinatorial optimization, recognized for being NP-complete and its applications across chemistry, network analysis, and pattern recognition. State-of-the-art methods, such as the McSplit algorithm and its successors, employ a recursive branch-and-bound procedure to navigate the vast solution space. The efficiency of this search is critically dependent on the initial vertex sorting heuristic, which not only guides the algorithm toward a good solution but also structures the search tree for the computationally intensive proof of optimality. The original algorithm relies on a simple node degree heuristic, which is often suboptimal. This paper systematically investigates the influence of alternative vertex-ordering heuristics on McSplitDAL, a state-of-the-art variant of McSplit. We integrate five node-ranking heuristics (namely, PageRank, Betweenness Centrality, Closeness Centrality, Local Clustering Coefficient, and a modified Katz Centrality) into the McSplitDAL framework. We analyze their effect on search-space exploration, pruning efficiency, convergence behavior, and execution speed. We also investigate how they shape the algorithmic search and affect the solver's ability to approach or prove optimality under constrained computa-

Andrea Calabrese, Lorenzo Cardone, Salvatore Licata, Marco Porro, and Stefano Quer contributed equally to this work.

✉ Lorenzo Cardone
lorenzo.cardone@polito.it

Andrea Calabrese
andrea.calabrese@polito.it

Salvatore Licata
salvatore.licata@studenti.polito.it

Marco Porro
marco.porro@studenti.polito.it

Stefano Quer
stefano.quer@polito.it

[1] DAUIN - Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino 10129, Italy

Published online: 01 December 2025

tional budgets. Experimental results across heterogeneous datasets reveal that specific heuristics, such as PageRank and Katz Centrality, consistently promote more effective pruning and higher-quality intermediate solutions, offering valuable insights into the relationship between graph topology-derived measures and branch-and-bound performance.

Keywords Graphs · Algorithms · Heuristics · Maximum Common Subgraphs · Software

1 Introduction

Graphs provide a powerful abstraction for modeling relationships among entities in domains 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).

Within this context, the Maximum Common Induced Subgraph (MCIS) problem seeks to identify the largest subgraph that is shared by two graphs. Despite being well studied in the scientific literature since the 70s (Bron and Kerbosch 1973; Barrow and Burstall 1976), and even considering its conceptual simplicity, MCIS remains NP-complete (Garey and Johnson 1979), making it computationally demanding even for moderately sized instances. Consequently, advances in this field often rely on algorithmic refinements that improve the "efficiency of search" or the "strength of pruning," rather than altering the fundamental computational complexity of the task.

One of the most prominent exact algorithms for this problem family is McSplit, introduced in 2017 by McCreesh et al. (2017). This algorithm is a recursive branch-and-bound method that explores vertex correspondences between two graphs while maintaining an upper bound on potential solution size. The core idea is to label all vertices based on the presence of edges toward already selected nodes. In any position of the recursion tree, McSplit can compute an upper bound on the size of the local branch and prune it when redundant. Numerous extensions, such as McSplitRL (Liu et al. 2020), McSplitLL (Zhou et al. 2022), and McSplitDAL (Liu et al. 2022), have refined this approach. McSplitRL employs a Reinforcement Learning approach to refine the order of vertex selection based on knowledge gathered during exploration. McSplitLL, based on McSplitRL, outperforms its predecessor by utilizing a technique called Long Short-Term Memory, which addresses nodes with specific characteristics. McSplitDAL is built upon McSplitLL and introduces a technique called Dynamic Action Learning, improvíng the reward function of McSplitRL. Nonetheless, these algorithms still depend critically on a "vertex-ordering heuristic," which determines the sequence in which potential vertex pairs are examined. In McSplit and most of its variants, this ordering is based on vertex degree, i.e., a simple yet often suboptimal choice, as it may fail to capture deeper structural information within the graphs.

The objective of this work is not only to show the faster convergence and reduced runtime of our approach, but also to analyze how alternative heuristics influence the

structure and efficiency of the search process itself. Specifically, we aim to extend our previous research (Calabrese et al. 2023), which improved McSplitDAL with an additional heuristic, PageRank, by investigating how different notions of node ranking (widely studied in graph theory) affect the sequence of explored states, the distribution of pruning events, and the solver's trajectory toward optimal or near-optimal solutions. Understanding these effects is crucial, as it reveals which structural graph properties most strongly correlate with efficient search behavior, and it provides guidance for designing heuristics that balance exploration and pruning.

Building on the McSplitDAL framework, we propose five new variants (collectively denoted McSplitDAL+X), each incorporating one of the following heuristics: PageRank, Betweenness Centrality, Closeness Centrality, Local Clustering Coefficient, and a modified Katz Centrality. These metrics were selected for their complementary perspectives on node importance and connectivity, enabling a controlled analysis of how local and global graph properties influence the algorithm's branching and bounding decisions.

Our experimental evaluation encompasses both synthetic and real-world datasets, which vary widely in terms of size and connectivity. Rather than measuring success solely in terms of runtime, we assess search-space reduction, solution quality under time constraints, and heuristic stability across problem classes. This empirical perspective offers new insights into the role of vertex-ordering heuristics in branch-and-bound solvers, highlighting the trade-off between structural sensitivity and computational overhead. Moreover, our experiments reveal that specific guides, such as PageRank and Katz Centrality, reliably enhance the performance of branch-and-bound algorithms by helping to prune the search more effectively and find good partial answers faster.

In summary, this work systematically evaluates vertex-ordering heuristics within McSplit and its variant. It presents an empirical analysis of how different centrality-based measures affect pruning behavior and search dynamics. Ultimately, it provides valuable insights into the design of heuristic strategies that can enhance the adaptability and theoretical interpretability of exact MCIS algorithms.

The paper is organized as follows. In Section 2, we describe our notation and define the problem. We also present an overview of the current relevant approaches for addressing this issue. In Section 3, we illustrate the newly proposed heuristics to enhance the McSplitDAL algorithm. Section 4 describes our testing setup, the datasets used, and the experimental results. Within this section, Subsection 4.5 reports an in-depth investigation to elucidate the effects of our heuristics on algorithmic performance. Finally, Section 5 draws some conclusions and hints at possible future work.

2 Background

This section introduces our graph notation, 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 benchmark baseline 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 describing 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 $E(G)$ $[E(H)]$ to represent the set of all 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 be categorized into several types: labeled or unlabeled, weighted or unweighted, and 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 $W(\{v_1, v_2\})$ associated with them. For example, a weight might represent the distance between two nodes in a topological graph. Unweighted graphs can be seen as weighted graphs with a unitary weight if the edge exists, and zero otherwise.

$$W(\{v_1, v_2\}) = \begin{cases} 1 & \forall \{v_1, v_2\} \in E(G) \\ 0 & forall \{v_1, v_2\} \notin E(G) \end{cases}$$

We say that G is *undirected* if:

$$\forall v_1, v_2 \in V(G), \{v_1, v_2\} \in E(G) \Rightarrow \{v_2, v_1\} \in E(G)$$
$$W(\{v_1, v_2\}) = W(\{v_2, v_1\})$$

In other words, if a link exists between v_1 and v_2, the opposite link must exist and have the same weight. In such cases, the two vertices are considered to be linked by a single bidirectional edge.

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

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

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 $f : V(G) \rightarrow V(H)$ between the vertices of two graphs G and H such that

$$\forall v_1, v_2 \in G \mid \{v_1, v_2\} \in E(G) \Rightarrow \{f(v_1), f(v_2)\} \in E(H)$$

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, as will the terms edge and link.

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 a subgraph of 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).

We focus on the Maximum Common Induced Subgraph problem between undirected, unlabeled, and unweighted graphs, representing the most generic and worst-case scenario for the MCIS 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 second graphs) that have the same connections toward vertices belonging to the current solution. As McSplit uses labels to identify possible couplings between vertices, the original algorithm also provides a method for creating those labels based on the adjacency lists of the vertices.

Algorithm 1 The simplified version of the original McSplit algorithm.

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

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 equal to or lower 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 continues to improve the current solution. The bound is computed as shown in Equation 1.

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

When improving the current solution, McSplit tries to build a more extensive solution by virtually removing a couple of vertices with the same label from the respective graphs and updating the remaining labels (lines 18-19) depending on the adjacency to the last selected node. Then, it tries to explore recursively all possibilities of further pairings 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 $v \in G$ and $w \in H$ are selected, the current mcs instance uses recursion to explore all solutions that include v,w, and all the nodes of the received partial solution M. Finally, 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 used 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., $max(|label(G)|, |label(H)|)$. 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 each vertex, choosing the one with the highest degree each time and removing it from the graph. We will refer to this approach as the *Node Degree*, or simply the *Degree* heuristic.

2.3 McSplit variants

Several notable variants of McSplit have been developed to enhance the original algorithm. This section briefly describes some of the most noticeable and recent ones. To be consistent with the notation, we name G the first graph passed as an input and H the second.

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 the denser graph as the first graph in 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

$$|\tfrac{1}{2} - K(G)| > |\tfrac{1}{2} - K(H)|$$

is true.

2.3.2 McSplitRL

Liu et al. (2020) propose McSplitRL, a novel approach that extends the standard McSplit using Reinforcement Learning. This approach maintains two vectors, one for the vertices of G and the other for the vertices of H, which contain the rewards associated with 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:

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

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 causes a higher reduction of the bound, therefore cutting as many branches as possible in subsequent steps of the algorithm.

2.3.3 McSplitLL

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-term Memory (LSTM) 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 substantial, an asymmetric decay is used, following a long-term and 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 pairs 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 that it can always be added to the current subgraph if its only neighbor is part of it. Thus, whenever a leaf from the left graph and a leaf from the right graph are found, the pair formed by these two nodes is automatically added to the current solution without needing new expensive recursive steps.

2.3.4 McSplitDAL

Liu et al. introduce McSplitDAL (Liu et al. 2022), built upon McSplitRL and McSplitLL, as a new improvement over the previous variants. 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. When branching, the DAL value function aims to consider reducing the upper bound and simplifying the problem occurring after the branch. This feature can be implemented by adding a 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:

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

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 for a dynamic strategy to select 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 (1973) casts the MCS problem onto the Maximum Common Clique problem. McCreesh et al. (2016) and Vismara and Valery (2008) follow the previous approach while exploiting constraint programming to solve the problem. Other approaches take a step back, adopting the parallel computation capabilities of General-Purpose computing on Graphics Processing Units (GPGPUs) (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 and Quer (2023). However, the most promising heuristics work by analyzing graphs in pairs and then merging the results, thus still motivating their research on MCS techniques that work on pairs of graphs.

3 Our Approach

The primary objective of this work is to enhance 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 Martí and Reinelt (2022):

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

Our heuristics also aim to generate diverse classifications 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 single integers, and the results may depend on the lexicographical order of the vertices. Considering these facts, we focus on classifying vertices based on single numbers. In particular, we developed five variants of McSplitDAL (combined with McSplitSD) using five different heuristics for classifying vertices:

- A variant using PageRank.
- A variant using Closeness Centrality.
- A variant using Local Clustering Coefficient.
- A variant using Betweenness Centrality.
- A variant using a modified version of Katz Centrality.

3.1 PageRank (PR)

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 rank the results of its web search engine. However, it is no longer used, as its patent expired in 2019.

PageRank is typically implemented on a generic graph, considering both directed and unweighted graphs to account for the diverse web pages. 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, which 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 and Page (1998)'s recommendation for the value of DF, setting it to 0.85. In line 2, we set the acceptable error ϵ at an arbitrary value. Experimentally, we find that the smaller the epsilon (i.e., the greater the precision of the procedure), the better the results, as the rankings tend to be more diverse. However, as the original algorithm accepts integer numbers, we also want to be able to map integers to ranks; thus, we chose ϵ to be 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 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 the 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 flows through each adjacent edge. In line 18, we transpose the stochastic matrix, replacing outgoing links with incoming links and vice versa. PageRank ranks nodes based on their incoming links; therefore, the inversion is necessary for the algorithm's generality. For undirected graphs, this might represent an unnecessary step; however, as McSplit works on directed and undirected graphs, this must also be true 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 the clicking probability at each iteration, 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 $(error < \epsilon)$ in line 25; this condition is triggered when the rankings converge and reach a stable configuration.

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

3.2 Closeness Centrality (CC)

The Closeness Centrality of a vertex evaluates its centrality inside a given graph and was first introduced by Bavelas (1950). This indicator can be calculated as the reciprocal sum of the lengths of all the shortest paths $d(v_i, v_j)$ originating from the candidate node v_i. It is worth noting that a higher score implies higher centrality, because a central vertex has the smallest sum of shortest distances, thereby having a higher closeness coefficient. Equation 5 formalizes the formula for computing this

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

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

$DF \leftarrow 0.85$
$\epsilon \leftarrow 0.00001$

Function *StochasticGraph*(G, *out_links*)
$G_s \leftarrow [0.0] * |G|$
for all $x, y \in adj(G)$ **do**
  if ($out\_link[x] = 0$) **then**
    $G_s[x, y] \leftarrow \frac{1.0}{|G|}$
  else
    $G_s[x, y] \leftarrow \frac{G[x,y]}{out\_link[x]}$
  end if
end for
return G_s

Function *PageRank*(G)
out_links $\leftarrow$ *OutLinksForEachNode*(G)
$G_s \leftarrow$ *StochasticGraph*(G, *out_links*)
$G_t \leftarrow$ *TransposeMatrix*(G_s)
result $\leftarrow \emptyset * |G|$
$p \leftarrow \emptyset$
for all $x, y \in adj(G_t)$ **do**
  push($\frac{G_t[x,y]}{|G|}$)
end for
error $\leftarrow 1.0$
while *error* $> \epsilon$ **do**
  result $\leftarrow \emptyset * |G|$
  for all $x, y \in adj(G_t)$ **do**
    $result[x] \leftarrow result[x] + G_t[x, y] * p[y]$
  end for
  for all *rank* $\in$ *result* **do**
    $rank \leftarrow rank * DF + \frac{1.0-DF}{|G|}$
  end for
  error $\leftarrow 0.0$
  for all *rank*, *prev* $\in$ *zip*(*results*, p) **do**
    $error \leftarrow error + abs(rank - prev)$
  end for
  $p = result$
end while
return result

score.

$$CC(v_i) = \frac{1}{\sum_{v_j \in G} d(v_i, v_j)} \tag{5}$$

In other applications, the CC is normalized by a factor of $|G| - 1$, but this linear correction is irrelevant to our application as a priority score generator.

3.3 Local Clustering Coefficient (LCC)

The Local Clustering Coefficient was introduced by Watts and Strogatz (1998), and is accepted as a measure of the closeness of the neighborhood of a given vertex to a complete graph (every node has a connection with all the others).

This measure can be calculated by counting, for a given candidate node vertex v, the number of its neighbors $v_j \in N_v$ and then counting the number of edges $e_jk \in E$ that connect those neighbors. Finally, the ratio between the neighborhood's connections over the total number $\frac{|N_v|\cdot(|N_v|-1)}{2}$ of possible links between all the adjacent vertices in the N_v neighborhood will output a score between 0 and 1, indicating the LCC for v. If $LCC(v) = 0$, then v is an isolated node, if $LCC(v) = 1$ then $N_v \cup \{v\}$ is a clique.

Equation 6 formalizes it.

$$LCC(v) = 2\frac{|e_{jk} : v_j, v_k \in N_v, v_j \neq v_i, e_{jk} \in E|}{|N_v|\cdot(|N_v|-1)} \tag{6}$$

LCC represents a loose indication of centrality, as nodes at the edge tend to be leaves or poorly connected vertices. By prioritizing nodes in well-connected neighborhoods, LCC is a promising candidate for a heuristic that tackles the most difficult branching decisions first in the recursion tree, speeding up the search through pruning. Leaf nodes have an LCC of 1, so they have the maximum priority and could slow down the search in the older McSplit or McSplitRL algorithms. However, it plays nicely with the LUM property of McSplitLL and McSplitDAL, which automatically and efficiently handles these edge cases.

3.4 Betweenness Centrality (BC)

Betweenness Centrality is a relevant index in graph theory, first introduced by Freeman (1977). This measure considers how many shortest paths pass through a given vertex v among all the shortest paths between any pair $(s, t) \in G$. If one of these node pairs has more than one shortest path traversing through v, only one is considered. It is beneficial in a wide range of applications because it expresses the quantity of information passing through a given vertex within a network, which is of utmost importance in fields such as telecommunications.

It is noteworthy to notice that even if it involves the shortest distances, it is different from Closeness Centrality since a *central* vertex, according to the betweenness criterion, must be included in the shortest paths, not having small shortest paths leading to other vertices (simply being an endpoint). The implementation we used follows the formalization expressed by the following Equation 7:

$$BC(v) = \sum_{s,v,t \in G, s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{7}$$

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

3.5 Katz Centrality* (KC)

The last heuristic we will consider is called Katz Centrality, introduced by Katz (1953) in the 50s. For any $k \in \mathbb{N}$, this heuristic counts the number of vertices in the graph that can reach the candidate vertex v by a walk of k hops. The metric computes an average of these counts, weighted by a penalty factor α with exponential decay for distant neighbors.

A formalization of Katz Centrality is Equation 8:

$$KC(v) = \sum_{k=1}^{\inf} \sum_{j=1}^{n} \alpha^k (A^k)_{vj} \tag{8}$$

where matrix A^k indicates whether two vertices are connected in k walks, and n is the number of graph nodes.

However, due to the high cost of calculating this metric, we used a simplified version that considers only the shortest walks σ_{vj}, rather than all possible walks of length k. To make the distinction clear, from now on, we will use the term "Katz Centrality*".

3.6 McSplitDAL+X

Within the framework introduced in Section 3.1, we exploit the ideas presented by McSplitDAL, enhanced by integrating the newly proposed heuristic node priority orders. The union of these techniques produced five new versions of the McSplitDAL algorithm, collectively referred to as McSplitDAL+X.

While the original McSplit idea was centered on the node degree heuristic, the subsequent variants were primarily based on McSplitRL, which employed reinforcement learning as a dynamic vertex selection policy. However, the node selection algorithm returns to the static heuristic whenever a tie is encountered. It is worth noting that relations are frequent in the initial stages of the search since all RL rewards are initialized to zero. Therefore, the heuristic holds considerable importance in our non-exhaustive recursive searches.

Algorithm 3 A template for the proposed McSplitDAL+X algorithm using McSplitDAL with a specific heuristic sort order produced by the `X` function.

Function (G, H)
$G_{ranks} \leftarrow X(G)$
$H_{ranks} \leftarrow X(H)$
$G_{sorted} \leftarrow SortGraph(G, G_{ranks})$
$H_{sorted} \leftarrow SortGraph(H, H_{ranks})$
$McSplitDAL(G_{sorted}, H_{sorted})$
return

The application of all heuristics is the same: a given scoring algorithm `X(G)` produces scalar scores for every single node of a graph, where nodes with high scores should be considered first. These scores are used to sort the vertices of the graphs, which are represented through an adjacency list. This approach is summarized by

Algorithm 3. First, we apply the heuristic to classify the vertices of graphs G and H (in lines 2 and 3, respectively). Then, both graphs are sorted using the corresponding scores (lines 4 and 5). Finally, McSplitDAL is run on the sorted graphs (line 6). Within McSplitDAL, nodes are selected based on the RL policy, and in the event of ties, the first node is chosen to ensure efficiency during the recursive steps of the algorithm. The proposed structure enables the heuristic algorithms to be computed only once, thereby reducing their computational costs to a small or negligible level.

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 compiled with GCC version 9.4. For McSplit, we use the original versions obtained from the web and adapted for use with our new heuristic. For McSplitDAL, we wrote an implementation that follows the ideas presented by the authors (Liu et al. 2022), as we could not find an official version publicly available. In addition, since it has been proven beneficial, we borrow the graph swap idea from McSplitSD (Trimble 2023) and incorporate it into all variants of McSplit.

Our tests are designed to evaluate the most practical aspects of all algorithms; thus, we assess their ability to find suitable solutions within a limited time frame, rather than 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 ability to find the most effective solution within the given time frame.

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, improving it only sporadically. Figure 1 plots the typical growth of the solution size for a variable number of recursions. We can observe that, at the beginning (within a few thousand recursions, typically completed in under one second in our setup), the solution size increases 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 note that the number of recursions is reported on the x-axis on a logarithmic scale.

4.2 Datasets

We tested all algorithms on two different datasets, which we will refer to as SMALL and LARGE. We focused on the most extensive graphs, the ones with the most nodes. Due to the massive size of the datasets, we selected a subset of graphs on which we ran our tests.

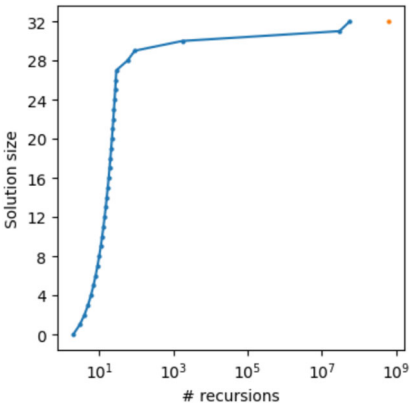


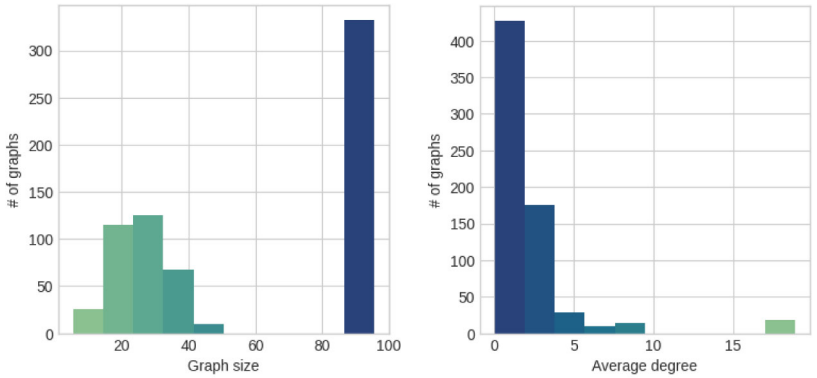
Fig. 1 Typical behavior of the effectiveness of the original implementation of McSplit. The size of the solution often increases rapidly in the initial part of the process; then, the procedure becomes trapped in local minima, which slow down the convergence process and force the algorithm to explore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

1. SMALL: A collection of graph pairs from a public MCS dataset (Foggia et al. 2001). The original dataset comprises 54,600 small graph pairs with diverse characteristics, from which we sampled 675 graph pairs uniformly distributed across all generation strategies, all composed of 100 nodes or fewer. This dataset is relatively homogeneous in size, but its difficulty varies significantly. The connectivity ranges from 10% to 90%. This dataset tests the algorithms on numerous smaller graph pairs to obtain a more comprehensive statistical overview of their performance. Figure 2a shows the actual size distribution of the graph pairs, and it shows that some minor pairs were added to detect eventual outlier behaviors in case of trivial problems. Figure 2b shows the distribution of the average degree of the dataset graphs, showing that various graphs with different properties have been included.
2. LARGE: A collection of graph pairs derived from the AS-733 dataset, collected from Stanford University (Leskovec and Krevl 2024). 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 500 and 6,000 nodes. These graphs were sorted by size and then paired to achieve 366 pairs with the minimum possible size ratio. The distribution of the graph size, reported in Figure 3a, follows a relatively broad distribution, with most graphs having a size between 3,000 and 5,000 nodes, which will have to be considered during the analysis. The average node degree hovers around 1.5, with less overall variation (Figure 3b). This dataset tests the algorithms on large, heterogeneous real-world graph pairs to evaluate their performance on more challenging and varied problems.

4.3 Comparison metrics

Two different plots will be used to compare the different heuristics:

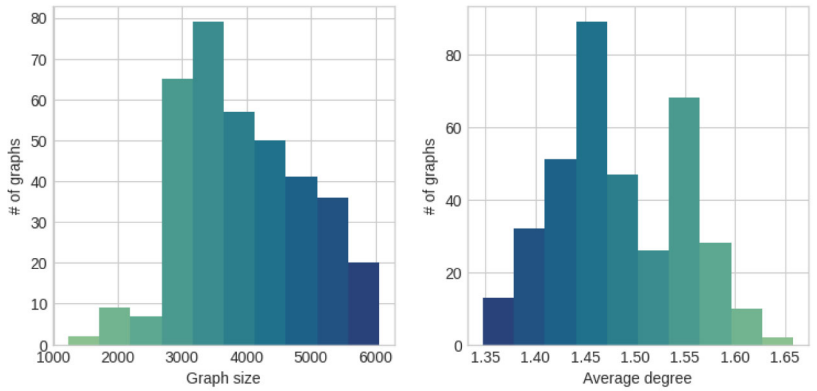
1. **Gain Plots**: Given a solver, the graph shows the solution size for each test instance, i.e., for each pair of graphs in the dataset. On top of this data are applied a few post-processing steps in sequence:



(a) Size distribution of the graph pairs, computed as the average size of the two graphs.

(b) Average degree distribution.

Fig. 2 Graph size distribution of the SMALL dataset



(a) Size distribution of the graph pairs, computed as the average size of the two graphs.

(b) Average degree distribution.

Fig. 3 Statistics of the LARGE dataset

(a) Sorting of the instances from smaller to larger solution size. When multiple solvers are compared, the sort is done relative to the average solution size produced by all solvers.

(b) Normalization. For each instance, all data is divided by the solution size of a specified method or by the average of all methods. This strategy helps to have a more precise visualization and highlights the relative performance independently of the graph size.

(c) Rolling Average. The solution size can vary from instance to instance, making the plots difficult to understand. Since the interest of the analysis is to detect the general trends, we average the results over windows of instances. In practice,

assuming a set of N MCS solutions S and a 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 W results.

$$r_i = \frac{1}{W} \sum_{j=i}^{i+W} |S_j| \qquad \forall i \in [1, N - W] \tag{9}$$

The goal of the gain plots is to visually detect patterns and differences in the behavior of the analyzed heuristics.

2. **Mean Normalized Difference (MND)**: While the gain plot visually represents the data, MND is used to obtain a numeric qualifier for the relative performance of two solvers. It is computed as the average difference between the solution sizes of two algorithms, normalized by the average of their solution sizes. Formally, given two distributions of N values, A and B, the MND is defined as in Equation 10:

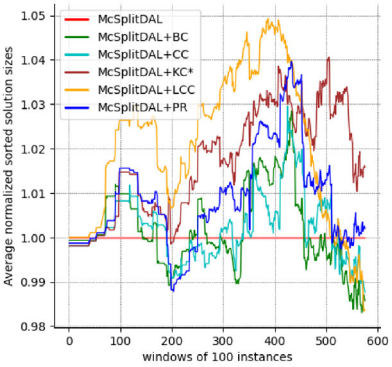
$$\text{MND}(A, B) = \frac{100}{N} \sum_{i=1}^{N} \frac{A_i - B_i}{\frac{A_i + B_i}{2}} = \frac{100}{N} \sum_{i=1}^{N} 2\frac{A_i - B_i}{A_i + B_i} \tag{10}$$

The formula multiplies the result by 100 to obtain a percentage value. This metric is used because solution sizes are highly variable; therefore, a regular average of differences would be skewed towards instances with large solution sizes. 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.

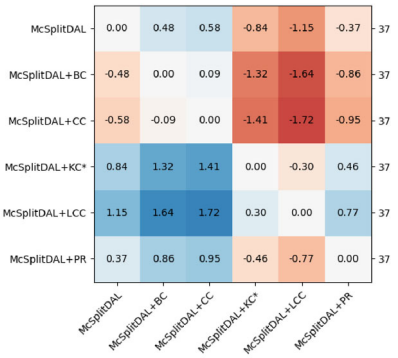
4.4 Experimental evaluation

In Fig. 4a, we show the main plot of the average performance of our McSplitDAL-normalized heuristics on the SMALL dataset. Due to normalization, McSplitDAL always has a value of one, whereas all other methods display values that vary depending on their relative performance compared to the baseline. Despite using the circular average, the algorithms present a considerable variation within the test suite. However, some general patterns can be detected. The solution sizes of the 50 smallest graphs are represented in the left portion of the plot, where all solvers show comparable performance (within 1%). The result is expected, as the problem is easy to solve, and all methods can reach a close-to-optimal solution in the given timeout.

As the remaining instances have all the same size but different connectivity, the complexity of the problem can vary. Since more difficult problems lead to smaller solution sizes, the solution size can be considered a rough proxy metric for the instance complexity. Following this reasoning, we can postulate that all new heuristics perform better for more challenging graph pairs. In the case of more straightforward problems (right side of the plot), the relative difference shrinks because all methods manage to get reasonable solutions of size greater than 80. Notably, Katz Centrality demonstrates a relative advantage over the other heuristics on complex problems, while the Local Clustering Coefficient performs well across the spectrum.



(a) Gain plot with a window width of 100 consecutive tests of the sizes of the solutions obtained by the McSplitDAL+X algorithms. All values are normalized with respect to the results obtained by McSplitDAL.



(b) Heatmap of the MND between solvers.

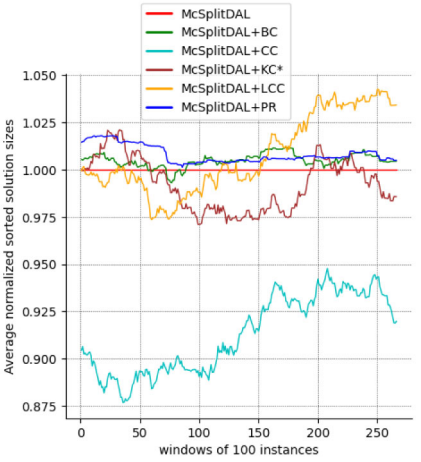
Fig. 4 Performance of McSplitDAL+X over the SMALL dataset

Fig. 4b instead shows a heatmap of the MND between the different heuristics applied to McSplitDAL. This plot does not indicate the behavior of a specific class of tests, such as the gain plot, because it averages over the entire set of tests. However, this method allows us to estimate, on average, whether a methodology is suitable for the whole of the test suite or not. As we can see, we have three heuristics that can outperform McSplitDAL using the degree heuristic: Katz Centrality*, Local Clustering Coefficient, and PageRank. This perspective confirms our previous claims, as shown in the gain plot.

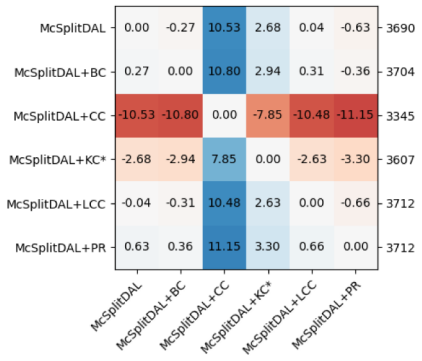
Regarding the LARGE dataset, we present our results in Figures 5a and 5b. This dataset exhibits high variability in terms of graph size, and solving the MCS problem is almost always impossible due to the task's complexity. Therefore, this test suite is instrumental in differentiating heuristics when McSplit's main goal is to converge to a perfect solution quickly due to the intractability of finding the optimal one.

The gain plot remains fundamental in distinguishing between the behavior on different types of tests. More specifically, as the results are ordered by increasing solution size, the more complex problems are located on the rightmost part of the plot. We can see that only one heuristic seems to have degraded performance for the other: Closeness Centrality. Once again, PageRank appears to be pretty stable across different test classes; however, in this case, Betweenness Centrality also delivers excellent overall performance. The local clustering coefficient yields outstanding results with more significant input graphs. At the same time, Katz Centrality* seems less efficient than what we observed in the small dataset.

Finally, we can see a recap of the various methods using the MND heatmap, which quantifies the improvement in the performance of the best methods for this dataset, namely PageRank and Betweenness Centrality, with respective scores of 0.63% and



(a) Gain plot with a window width of 100 consecutive tests of the sizes of the solutions obtained by the McSpliDAL+X algorithms. All values are normalized to the results obtained by McSplitDAL.



(b) Heatmap of the MND between solvers.

Fig. 5 Performance of McSplitDAL+X over the LARGE dataset

0.27%. It should be noted that these values are lower than those observed on the SMALL dataset, as the MND metric behaves as a linear relative comparison, despite the problem complexity increasing exponentially with the size of the input graphs. On the other hand, the worst-performing heuristic (the Closeness Centrality) manages an MND score of -7.85% relative to the original degree-based sort order.

In Fig. 6, we comprehensively compare the solution sizes achieved by each McSplitDAL+X method for the original degree sort order. For each instance, a dot is reported to show the size of the solutions found by the two algorithms on the SMALL dataset. By eliminating the need for a rolling average, this scatter plot provides a clearer view of the results for individual instances. The considered charts show that the McSplitDAL+LCC and McSplitDAL+PR perform better than McSplitDAL because they deliver more extensive solutions more often.

4.5 In depth analysis

In this section, we undertake a thorough investigation of our key heuristics to elucidate their distinct effects on algorithmic performance. Our analysis particularly focuses on the efficacy of heuristics in scenarios where the algorithm successfully terminates within the predefined timeout, thereby clarifying their contributions to the efficiency of problem resolution under favorable conditions. We analyze the behavior of the proposed heuristics in detail as the ordering of nodes changes with respect to the considered metric. In particular, we examine the following three orderings, which we will refer to in the rest of the section:

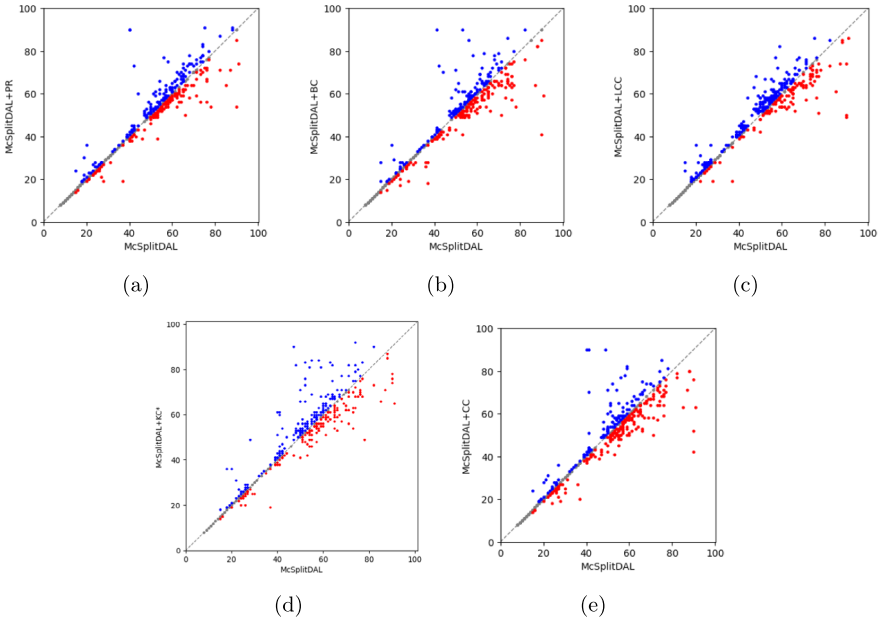


Fig. 6 XY plots of the sizes of the solutions found by each heuristic concerning the original degree sort order on the SMALL dataset. The dispersion of the points above the main diagonal (in blue) shows the instances for which the considered method finds solutions larger than the baseline. Grey points on the diagonal represent instances where both methods find a solution of the same size. Red points below the main diagonal show instances where the considered method finds a smaller solution than the baseline

- The "standard" (or "forward") order indicates the original ordering, as initially designed by each heuristic.
- The "reverse" (or "backward") order denotes that the nodes of both graphs are processed in the reverse order with respect to the standard case.
- The "inverse" order shows a configuration in which only the nodes of one of the two graphs are sorted in reverse order with respect to the standard case.

All values we report in this section are obtained by averaging the results over a large set of more than 200 graph pairs, including only those for which the optimality of the solution was confirmed within the predetermined time limit. Where relevant, we report the aggregate data, providing a general view of the algorithm's behavior on generic graphs, regardless of their specific characteristics. We then divide these 200 graph pairs into two groups of about 100 pairs each, depending on whether the McSplitDAL algorithm, using the degree-based heuristic, found the optimal solution faster in standard (forward) or reverse (backward) sorting. From here on, we will refer to the first group as **Std-Best** and the second one as **Rev-Best** to easily distinguish which portion of the tests we are addressing. Although the two subgroups are fairly balanced (with a slight predominance of the group in which the backward sorting is more efficient), it is observed that about 68% of the total number of iterations is spent trying to solve the pairs of graphs belonging to the latter group. This imbalance suggests that, on average, the number of iterations per graph pair is significantly higher in cases where

reverse sorting produces better results. This result provides us with more detailed information about the algorithm's behavior and, as we shall see below, demonstrates how the algorithms can exhibit different performances. Consequently, it also suggests that being able to distinguish between these two graph categories in advance could offer significant advantages in terms of efficiency.

4.5.1 Optimum search and maximality proof

Since all heuristics consistently solve the considered instances within the timeout, we can scrutinize their trajectory toward optimality. The core idea is to quantify the number of vertices paired by each heuristic before finding an optimal solution. This investigation illuminates two previously unexamined dimensions of performance: The speed at which an algorithm converges to the optimal solution, and the computational effort invested in verifying that maximality. Unpacking these aspects is crucial to understanding the nuanced performance differences that distinguish these algorithms, providing insights that a simple success-within-timeout metric cannot offer.

Table 1 shows the number of iterations required for the different sorting algorithms to find the maximum solution (columns "MS") and to check its maximality (columns "MC"). All data are compared against the original version of the algorithm, which sorts all vertices by their degree in descending order. We can observe that the inverse version of each algorithm almost always produces worse results than the other two variants, confirming that sorting has a positive effect, reducing search time. On the contrary, the reverse ordering often produces better results, but this behavior is better analyzed by Tables 2 and 3.

Table 2 reveals a particularly insightful contrast by detailing instances where the Degree heuristic forward sorting outperforms its reverse sorting. A key observation is that, among all heuristics analyzed, only the reverse variant of Closeness Centrality achieves a reduction in total iterations compared to the standard forward-sorted Degree. This specific advantage aligns with previous findings that identify Closeness Centrality as being exceptionally resilient to the effects of graph splitting. Conversely, all other heuristics exhibit a significant decline in efficiency in the reverse mode, impacting both the iterations required to find the optimal solution and those subsequently needed to confirm its optimality.

Conversely, Table 3 illustrates the contrasting scenario in which the majority of the algorithms demonstrate enhanced performance in reverse mode. Notably, in numerous instances, the efficiency achieved by the reverse version of the heuristics exceeds the original ones based on the node's degree.

To better understand the previous analysis, we observe that when the forward order is superior, the reverse order significantly worsens the choice of node pairings, increasing the number of backtracks required to find the optimal solution. Our data also shows that for the reverse order, the bound decreases slightly slower on average than for the forward sorting, and the average depth achieved before pruning is greater. These three factors indicate that the algorithm based on reverse sorting explores deeper branches without improving the current solution. In contrast, in cases where the reverse sorting is more effective, we can observe that the number of pairings tested to find the optimal solution is significantly lower. In addition, the degree sorting is considerably less

Table 1 Aggregated results: Percentage of the total iterations with respect to the standard degree sorting. Smaller values are better

Sorting	Degree		Pagerank		Katz Centrality		Local Clustering Coefficient		Closeness Centrality		Betweenness Centrality	
	[%]		[%]		[%]		[%]		[%]		[%]	
	MS	MC	MS	MC	MS	MC	MS	MC	MS	MC	MS	MC
Standard	100.0	100.0	102.3	102.6	99.2	102.1	138.8	114.5	161.3	130.1	116.7	107.2
Reverse	90.3	89.2	77.5	83.8	50.5	71.0	175.8	132.6	50.4	70.1	70.3	81.0
Inverse	142.1	118.2	119.6	109.7	130.2	114.7	154.4	123.0	152.5	124.4	124.9	109.7

Table 2 Std-Best set: Percentage of the total iterations compared to the ones of the standard degree sorting. Smaller values are better

Sorting	Degree		Pagerank		Katz Centrality		Local Clustering Coefficient		Closeness Centrality		Betweenness Centrality	
	[%]		[%]		[%]		[%]		[%]		[%]	
	MS	MC	MS	MC	MS	MC	MS	MC	MS	MC	MS	MC
Standard	100.0	100.0	243.3	131.5	213.4	127.7	514.9	179.5	467.3	181.9	310.3	140.2
Reverse	373.1	144.0	347.2	137.2	201.4	105.6	551.3	187.9	160.6	96.9	335.0	136.8
Inverse	363.5	154.9	340.5	150.5	357.8	155.7	523.5	183.8	454.3	175.9	375.8	153.0

Table 3 Rev-Best set: Percentage of the total iterations compared to the ones of the standard degree sorting. Smaller values are better

Sorting	Degree		Pagerank		Katz Centrality		Local Clustering Coefficient		Closeness Centrality		Betweenness Centrality	
	[%]		[%]		[%]		[%]		[%]		[%]	
	MS	MC	MS	MC	MS	MC	MS	MC	MS	MC	MS	MC
Standard	100.0	100.0	77.1	89.2	78.9	90.3	71.8	84.5	106.7	106.2	82.2	91.9
Reverse	39.9	63.9	29.4	59.1	23.6	55.0	108.8	107.0	30.8	57.7	23.1	55.3
Inverse	102.7	101.3	80.3	90.9	89.6	95.8	88.6	94.9	98.7	100.7	80.1	89.7

Table 4 Percentage of the search space explored before finding the best solution on solved graphs

Sorting	Degree [%]	Pagerank [%]	Katz Centrality [%]	Local Clustering Coefficient [%]	Closeness Centrality [%]	Betweenness Centrality [%]
Standard	47.5	47.4	46.2	57.6	58.9	51.8
Reverse	48.1	43.9	33.8	63.0	34.2	41.2
Inverse	57.1	51.8	53.9	59.7	58.3	54.1

Table 5 Standard sorting: Percentage of the search space explored before finding the best solution on solved graphs

Sorting	Degree [%]	Pagerank [%]	Katz Centrality [%]	Local Clustering Coefficient [%]	Closeness Centrality [%]	Betweenness Centrality [%]
Standard	22.76	42.12	38.04	65.31	58.47	50.38
Reverse	58.98	57.59	43.43	66.80	37.72	55.74
Inverse	53.42	51.52	52.32	64.84	58.81	55.93

effective than the other heuristics, and the average depth reached before pruning is greater, suggesting that the search branches are pruned later.

We now consider a secondary, yet significant, insight that emerges from our results. As noted, the number of iterations required to find the optimal solution is not necessarily the same as the number needed to verify its optimality. Consequently, Tables 4, 5, and 6 report the percentage of the total search process that has been completed when the optimal solution is first identified. The computational effort beyond this point is dedicated to verifying that the discovered solution is indeed maximal. This data, which is broadly consistent with earlier analyses, effectively highlights how specific heuristics can lead to an optimal solution more quickly by exploring a relatively confined segment of the search space.

Table 4 shows a consistent trend across the various heuristics: The reverse ordering reaches the optimal solution while exploring the smallest portion of the search space, followed by the standard (forward) ordering, and lastly the inverse ordering. This behavior holds for all heuristics, including the degree sorting, with the notable exception of the Local Clustering Coefficient strategy, which exhibits a performance drop when its ordering is reversed.

Going into more details, Table 5 shows the cases where the forward ordering produces the best results under the degree heuristic. As expected, the row corresponding to this ordering tends to report the lowest average percentages.

Conversely, Table 6 shows the cases where the backward ordering produces the best results under the degree heuristic. In this case, the lowest percentages are generally associated with the reverse sorting. Moreover, several heuristics have a significant edge over the degree one.

It is also worth noting that some of the heuristics maintain the respective order of efficacy regardless of whether the degree heuristic performs better in the standard or

Table 6 Reverse sorting: Percentage of the search space explored before finding the best solution on solved graphs

Sorting	Degree [%]	Pagerank [%]	Katz Centrality [%]	Local Clustering Coefficient [%]	Closeness Centrality [%]	Betweenness Centrality [%]
Standard	58.98	51.00	51.53	50.13	59.27	52.73
Reverse	36.81	29.33	25.29	59.98	31.47	24.64
Inverse	59.77	52.08	55.16	55.05	57.82	52.66

reverse configuration. Interestingly enough, no heuristic showed an opposite behavior than the degree one, i.e., either the reverse sort improved the results in the standard best cases or the standard sort improved the reverse best cases. As a final remark, notice that although the dataset includes a large number of graphs, the observed percentages exhibit high variance across different experiments and should therefore be interpreted with caution. The general trend suggests that larger graphs tend to reach their maximal solutions earlier in terms of iteration count.

In essence, this section clarifies the circumstances under which the proposed heuristics achieve superior performance compared to the standard ordering, as well as the situations in which they fail to provide substantial benefits. Crucially, the differentiation between the Std-Best and Rev-Best sets of instances reveals two categories of problems with notably distinct computational characteristics. This analysis corroborates some of our earlier findings, but it also shows some discrepancies with the results presented in Section 4.4. For example, the Closeness Centrality and Betweenness Centrality heuristics (previously identified as less effective) continue to deliver the most underwhelming performance. Moreover, the three remaining heuristics exhibit more robust behavior, offering considerable improvements, particularly in critical instances where standard ordering falters, and demonstrate superior effectiveness on average. Conversely, the data presented in the previous section, including both the experiments discussed here and those terminated by the timeout, shows a significant similarity with the Rev-Best set (in particular with the row related to the "standard sort" strategy). Indeed, the Rev-Best instances seem to be the most significant drivers of the overall algorithm execution time presented in the previous section. These cases correspond to the scenarios where the original degree-based sorting yields its poorest results, being outperformed by almost all the other heuristics examined. This element suggests that a significant portion of the graphs not solved within the time limit belong to the Rev-Best category, while only a smaller fraction falls into the Std-Best category. In support of this hypothesis, we observe that, outside of the subset analyzed in this section, there are pairs of graphs for which only one of the two heuristics –standard degree sort and reverse degree sort –reaches the algorithm's conclusion within the 60-second timeout. Of these, most are solved by the reverse degree sort heuristic.

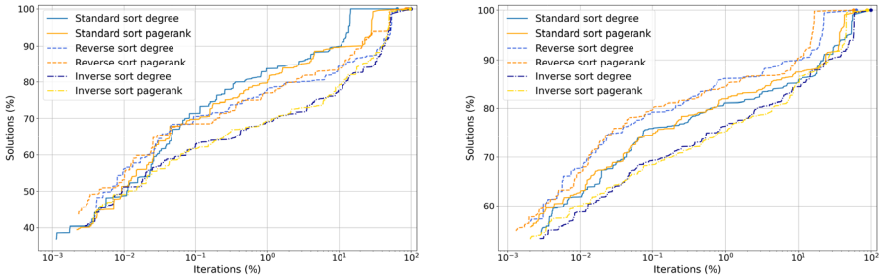
4.5.2 Execution profiles

This subsection employs graphical visualization to more closely examine how the solution evolves as a function of the iteration count. Additionally, it provides a more granular view of the heuristic behavior, drawing on an analysis of indicators such as the quantity of pruned branches, the mean search depth, and the average reduction in bounds following each matching process. Collectively, these details facilitate the identification of the most critical attributes a metric must possess to enhance the performance of the MCS search algorithm effectively.

For a consistent visual analysis, each figure in this section presents six performance curves. A set of three curves corresponds to the "degree ordering" (our standard reference throughout this section). The other three curves showcase the heuristic currently under evaluation, with each reflecting a different sorting order applied to it. These plots illustrate the percentage improvement of the solution against the progression of algorithm iterations for each sorting used by the heuristic. As in the previous section, the three curves represent the "standard" ("forward"), the "reverse" ("backward"), and the "inverse" ordering. All reported values are obtained as in the previous section, i.e., they are average values obtained considering the 200+ experiments that end within the selected timeout. The percentage of the solution achieved is self-explanatory. In contrast, the percentage of the iterations performed is computed with respect to the number of iterations required to guarantee the maximality of the final solution.

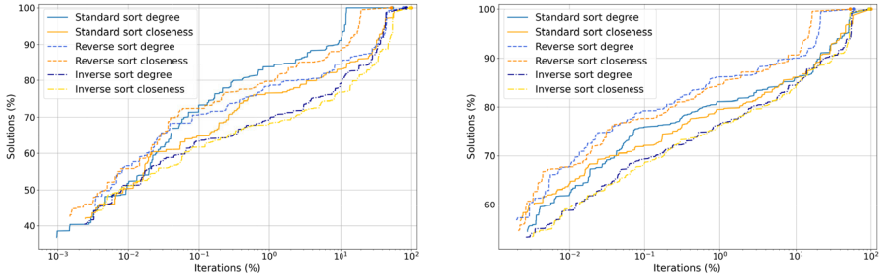
Figure 7 illustrates the current solution's size, expressed as a percentage of the final solution size, in relation to the number of graph node pairs examined, which is also reported as a percentage of its total final value. In scenarios where forward sorting proves most effective, we evaluate the Degree and PageRank heuristics based on their standard sorting methods. Consistent with previous tabular findings and confirmed by Figure 7a, sorting by the Degree heuristic demonstrates superior outcomes. In these instances, a slight decline in matching precision is observed, resulting in the traversal of a more extensive portion of the search tree. However, this particular behavior (more exhaustive tree exploration) is markedly less pronounced when using the reverse implementation of the Degree heuristic. Conversely, in cases where reverse sorting is the optimal approach, the most significant distinguishing factor is the quantity of branches pruned. This quantity is proportionally greater than the number of solutions tested, indicating a more efficient reduction of the search space and an overall improvement in the time required to reach a solution.

Figure 8 displays the same performance metric, this time specifically for the Closeness Centrality heuristic. In this analysis, we consistently examine its reverse version, as our experiments have shown it to be more effective on average. When forward sorting is the more advantageous strategy, the reverse Closeness Centrality initially shows lower matching accuracy. This behavior is comparable to that of the reverse Degree heuristic, although the effect is less pronounced with Closeness Centrality. Interestingly, once the optimal solution is attained using reverse Closeness, there is a significant increase in the average search depth achieved before pruning occurs, compared to the Degree heuristic. This observation suggests that the branches remaining at that stage potentially contain higher-quality solutions, thereby contributing to a reduction in the overall execution time. Conversely, in cases where reverse sort-



(a) Std-Best. (b) Rev-Best.

Fig. 7 Degree and Pagerank metrics solution size (%) plotted as a function of the iteration count (%)

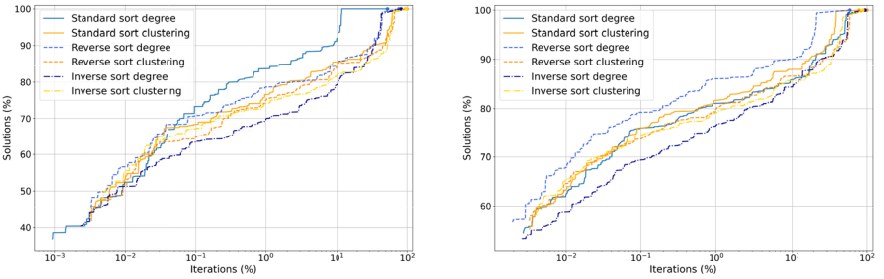


(a) Std-Best. (b) Rev-Best.

Fig. 8 Degree and Closeness Centrality metrics solution size (%) plotted as a function of the iteration count (%)

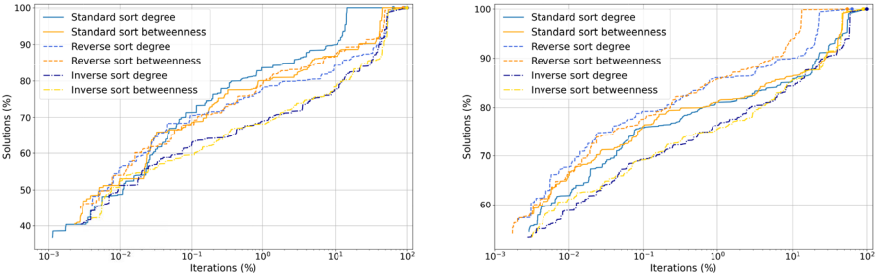
ing is most effective, the reverse Closeness Centrality heuristic improves upon the reverse Degree heuristic in two key aspects: it achieves greater accuracy in the initial matches and exhibits a proportionately higher rate of branch pruning. These factors combine to reduce the extent of the search space that needs to be explored, enabling more aggressive pruning and ultimately resulting in less time required to identify the optimal solution.

Figure 9 details the results for the Local Clustering Coefficient heuristic. In this instance, the general distinction between forward best and reverse best scenarios does not appear to alter the heuristic's effectiveness substantially; its forward version is consistently confirmed as the superior performer. A particularly noteworthy finding is that the forward version of this algorithm is the fastest among all those evaluated on generic graphs (refer to Table 1). This speed advantage is also partially evident in its reverse counterpart, even though reverse implementations generally incur greater computational costs. However, this competitive edge is not sustained when considering the reverse versions of other heuristics. When assessed in standard best cases (i.e., scenarios where forward sorting is typically optimal), this heuristic's sorting mechanism (its forward version) demonstrates inefficiency: the optimal solution is identified much later in the search progression and necessitates deeper exploration before backtracking can be effectively utilized. Conversely, in reverse best-case scenarios, the Local



(a) Std-Best. (b) Rev-Best.

Fig. 9 Degree and Local Clustering Coefficient metrics solution size (%) plotted as a function of the iteration count (%)



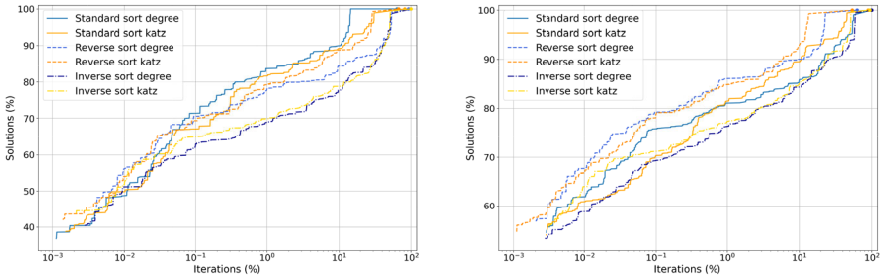
(a) Std-Best. (b) Rev-Best.

Fig. 10 Degree and Betweenness Centrality metrics Solution size (%) plotted as a function of the iteration count (%)

Clustering Coefficient heuristic generally finds the optimal solution more rapidly on average than the reverse Degree heuristic. Despite this, the algorithm tends to explore a larger volume of branches that are subsequently discarded, even with a shallower recursion depth, indicating a challenge in swiftly converging on the optimal solution.

Figure 10 illustrates the performance characteristics of the Betweenness Centrality heuristic. In situations where forward sorting proves most advantageous, the Betweenness heuristic typically fails to locate the optimal solution during the initial stages of the exploration process. Moreover, the segment of the search tree that is traversed under these conditions is often marked by numerous branches being cut at shallow depths, which implies that a substantial amount of time is likely spent pursuing unpromising solution paths. Conversely, when reverse sorting is the more beneficial approach, the algorithm demonstrates an ability to secure high-quality matches early on. This early success significantly narrows the breadth of the domain that needs to be explored. On average, such behavior results in more effective pruning mechanisms and greater overall efficiency during the exploration of lower recursion levels.

Lastly, Figure 11 presents the findings for the Katz heuristic, which emerges as another instance where reverse sorting methods consistently deliver the best performance. Similar to Closeness Centrality, it is the second heuristic observed whose



(a) Std-Best. (b) Rev-Best.

Fig. 11 Degree and Katz Centrality metrics solution size (%) plotted as a function of the iteration count (%)

optimal (reverse) version does not exhibit a substantial drop in effectiveness even under Std-Best conditions. Our experiments reveal that in Std-Best scenarios, the reverse Katz metric initially shows diminished efficacy in directing the algorithm toward the optimal solution during the early stages of the process. However, once the optimal solution has been identified, there is a marked increase in both the average depth of exploration and the magnitude of bound reduction achieved per pairing. This combination allows for a relatively rapid confirmation of the solution's maximality. In Rev-Best situations, while the general performance indicators for the reverse Katz heuristic are comparable to those of the reverse Degree version, the overall improved outcome with Katz is attributable to its higher sorting accuracy. This superior accuracy enables the algorithm to converge on the optimal solution more swiftly.

In summary, the heuristics demonstrating the highest performance were those that arranged nodes in a manner facilitating the relatively swift identification of the optimal solution within the search tree. Specifically, the most outstanding results were yielded by both the Betweenness Centrality and Katz Centrality metrics when their reverse ordering was utilized.

5 Conclusions and future works

This study revisits the Maximum Common Induced Subgraph (MCIS) problem through the lens of heuristic design and algorithmic behavior, rather than just focusing on runtime improvements. Building upon the McSplitDAL algorithm, we investigate how various vertex-ordering strategies (derived from established graph centrality measures) impact the structure and dynamics of the branch-and-bound search. Essentially, we look at the following five heuristic node prioritization schemes:

- PageRank (PR): This metric favors nodes that are more easily reachable through multiple connections across a network. It is widely recognized for its foundational role in Google's original search engine ranking system.
- Closeness Centrality (CC): This serves as an indicator of the average length of the shortest paths connecting a candidate node to all other nodes in the graph.

- Local Clustering Coefficient (LCC): This metric is used to estimate the "cliqueness" or density of a node's neighborhood. It is calculated as the ratio of existing interconnections among distinct neighbors compared to the maximum possible number of such connections.
- Betweenness Centrality (BC): This measures the frequency with which a candidate vertex lies on the shortest paths between any other pair of nodes within the network.
- Katz Centrality* (KC): This represents a weighted average of the shortest distances from a node to all other nodes, where the number of hops determines the weighting factor.

Our results show that vertex-ordering heuristics do not merely alter the order of exploration; they fundamentally impact how the algorithm prunes and prioritizes parts of the search space. In particular, specific heuristics, such as PageRank and Katz Centrality, consistently promote quicker convergence patterns, while others, like Betweenness Centrality, reveal their effectiveness only when the original degree heuristic fails. These findings emphasize that heuristic choice plays a pivotal role in the efficiency of the search process and the distribution of computational effort, even when all solvers ultimately guarantee the same optimality conditions.

From a methodological standpoint, the framework presented here enables a systematic evaluation of heuristic effects within exact MCIS solvers. Beyond their empirical performance, these heuristics act as probes into the internal mechanics of McSplit-like algorithms, exposing how different structural indicators of graphs influence the evolution of search and pruning boundaries. This understanding contributes to a broader perspective on the interplay between graph structure, algorithmic guidance, and proof of optimality.

In summary, this work contributes both empirical evidence and conceptual insights into how vertex-ordering heuristics influence the internal logic of branch-and-bound solvers for MCIS. By reframing heuristic design as a question of search-space reduction and proof efficiency, we aim to provide a clearer understanding of what makes specific heuristic strategies more effective.

Looking forward, several directions naturally emerge:

- Adaptive heuristic selection. Our findings suggest that the most effective heuristic depends on the graph's structural class. Future work could explore data-driven or meta-learning approaches to select or combine heuristics based on pre-computed graph descriptors automatically.
- Hybrid and dynamic ordering policies. Integrating multiple heuristics dynamically during the search may further improve pruning efficiency by adapting the ordering to the evolving state of the search tree.
- Parallel and distributed exploration. While this study focused on single-threaded execution for controlled comparison, the observed differences in heuristic-induced search patterns point to opportunities for parallelizing exploration across cores or nodes, guided by complementary heuristics.
- Theoretical analysis of heuristic effects. Ultimately, a more comprehensive theoretical understanding of how heuristic scores relate to expected pruning rates or bound tightness could provide a stronger foundation for designing practical heuristic functions in NP-complete search problems.

Funding Open access funding provided by Politecnico di Torino within the CRUI-CARE Agreement.

Data Availability Our code and all related experiments are available under GNU General Public License, on GitHub at https://github.com/Lorenzo-Cardone/McSplitDAL-PR-parallel or at https://github.com/stefanoquer/McSplitDAL-PR-parallel.

Declarations

Conflicts of Interest The authors declare no potential conflict of interest.

References

Angione, F. et al.: An innovative strategy to quickly grade functional test programs. Proceedings of the IEEE International Test Conference (ITC) 355–364 (2022)

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

Bavelas, A.: Communication patterns in task-oriented groups. The journal of the acoustical society of America **22**, 725–730 (1950)

Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. Computer Networks and ISDN Systems. Proceedings of the Seventh International World Wide Web Conference. **30**, 107–117 (1998). https://www.sciencedirect.com/science/article/pii/S016975529800110X

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

Calabrese, A., Cardone, L., Licata, S., Quer, S., Porro, M.: A web scraping algorithm to improve the computation of the maximum common subgraph. Proceedings of the 18^{18} International Conference of Software Technologies (ICSOFT 2023) 197–206 (2023)

Cardone, L., Quer, S.: The multi-maximum and quasi-maximum common subgraph problem. MDPI Computation **11** (2023). https://www.mdpi.com/2079-3197/11/4/69

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

Foggia, P., Sansone, C., Vento, M.: A database of graphs for isomorphism and sub-graph isomorphism benchmarking. Proc. of the 3rd IAPR TC-15 International Workshop on Graph-based Representations (2001)

Freeman, L. C.: A set of measures of centrality based on betweenness. Sociometry 35–41 (1977)

Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco, CA, USA (1979)

Katz, L.: A new status index derived from sociometric analysis. Psychometrika **18**, 39–43 (1953)

Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection (2014). http://snap.stanford.edu/data

Levi, G.: A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcolo **9**, 341–352 (1973). https://doi.org/10.1007/BF02575586

Liu, Y., Zhao, J., Li, C.-M., Jiang, H., He, K.: Hybrid learning with new value function for the maximum common subgraph problem (2022). arXiv:2208.08620

Liu, Y., Li, C.-M., Jiang, H., He, K.: A learning based branch and bound for maximum common subgraph related problems. Proceedings of the AAAI Conference on Artificial Intelligence **34**, 2392–2399 (2020). https://ojs.aaai.org/index.php/AAAI/article/view/5619

Martí, R., Reinelt, G.: Heuristic Methods, 27–57 (Springer, Berlin Heidelberg, Berlin. Heidelberg (2022). https://doi.org/10.1007/978-3-662-64877-3_2
McCreesh, C., Ndiaye, S. N., Prosser, P., Solnon, C.: Clique and constraint models for maximum common (connected) subgraph problems. Principles and Practice of Constraint Programming 350–368 (2016)
McCreesh, C., Prosser, P., Trimble, J.: A partitioning algorithm for maximum common subgraph problems. Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence IJCAI-17 712–719 (2017). https://doi.org/10.24963/ijcai.2017/99
Michael Garey, D.S.J.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, United States (1979)
Milgram, S.: The small world problem. Psychol. Today **2**, 60–67 (1967)
Park, Y., Reeves, D.: Deriving common malware behavior through graph clustering. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security 497–502 (2011)
Quer, S., Marcelli, A., Squillero, G.: The maximum common subgraph problem: A parallel and multi-engine approach. MDPI Computation **8** (2020). https://www.mdpi.com/2079-3197/8/2/48
Schöning, U.: Graph isomorphism is in the low hierarchy. Journal of Computer and System Sciences **37**, 312–323 (1988). https://www.sciencedirect.com/science/article/pii/0022000088900104
Trimble, J.: Partitioning algorithms for induced subgraph problems. Ph.D. thesis, University of Glasgow, Glasgow, UK (2023)
Vismara, P., Valery, B.: Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. Modelling, Computation and Optimization in Information Systems and Management Sciences: Second International Conference MCO 2008, Metz, France-Luxembourg, September 8-10, 2008. Proceedings 358–368 (2008)
Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**, 440–442 (1998)
Zhou, J., He, K., Zheng, J., Li, C.-M., Liu, Y.: A strengthened branch and bound algorithm for the maximum common (connected) subgraph problem (2022). arXiv:2201.06252
Zimmermann, T., Nagappan, N.: Predicting subsystem failures using dependency graph complexities. Proceedings of the 18th IEEE International Symposium on Software Reliability (ISSRE '07) 227–236 (2007)