universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Engineering Maximum Common Subgraph Algorithms for Large Graphs“

verfasst von / submitted by

Jonathan Trummer, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik
Betreut von / Supervisor:	Ass.-Prof. Dipl.-Inf. Dr. Nils Morten Kriege
Mitbetreut von / Co-Supervisor:	Dr. Kathrin Hanauer, B.Sc. M.SC. Univ.-Prof. PD. Dr. Christian Schulz

Acknowledgements

This work would not have been possible without the weekly feedback and guidance of my supervisors Ass.-Prof. Nils Kriege and Dr. Kathrin Hanauer as well as Prof. Christian Schulz. I am grateful for the work you have put in in helping me achieve this thesis.

Abstract

The *Maximum Common Subgraph* problem is to find the largest subgraph common to two given graphs. This problem finds applications in a wide variety of fields, for example it can be utilized to find common substructures between two molecules, which may be useful in the fields of chemistry, molecular science and computational drug discovery. As the problem is *NP-complete*, finding an optimal solution requires exhaustively exploring the solution space. Various different approaches have been applied to the Maximum Common Subgraph problem, such as reducing it to the *Maximum Clique* or *Minimum Vertex-Cover* problems, applying *Constraint-Satisfaction Programming* or utilizing a partitioning approach.

We propose two methods of data reduction, which may be used to reduce the sizes of the input graphs and can thus simplify the problem. Additionally, we propose two methods of computing upper bounds, as means to terminate the search for solutions as soon as the best solution found matches the upper bound. We explore the exploitation of symmetries, as to reduce the search space which needs to be explored, and we evaluate an *Independent Set* solver as an alternative solver. Finally, we extensively evaluate existing approaches and our techniques experimentally, where we show that a heuristic independent set solver may significantly outperform existing approaches both in terms of running time as well as result quality, given a maximum execution time per problem instance. We additionally introduce a pre-processing phase for a partitioning-based solver, which is able to reduce the running time on instances of one instance group by over 80%.

Kurzfassung

Das Problem des *Größten Gemeinsamen Subgraphen* sucht den größten Subgraphen, den zwei gegebene Graphen beinhalten. Dieses Problem lässt sich in vielen verschiedenen Forschungsfeldern anwenden, so kann damit zum Beispiel die größte gemeinsame Struktur zweier Moleküle bestimmt werden, was in den Bereichen der Chemie, Molekular Wissenschaften und Computer-gestützter Medikamenten-Entwicklung angewandt werden kann. Da das Problem *NP-vollständig* ist, lässt sich eine optimale Lösung nicht direkt berechnen, sondern der gesamte Lösungsraum muss abgesucht werden. Es wurden bereits verschiedene Ansätze angewandt, um Lösungen für das Problem des Größten Gemeinsamen Subgraphen zu finden. Dazu zählt eine Reduktion zum Problem der größten Clique, beziehungsweise zum Problem der kleinsten Knotenüberdeckung, sowie eine Formulierung als Bedingungserfüllungsproblem und Ansätze, die den Lösungsraum partitionieren.

Wir präsentieren zwei Methoden der Datenreduktion, welche genutzt werden können, um die Größe der Eingabe-Graphen zu reduzieren und somit die Lösungssuche zu vereinfachen. Weiteres präsentieren wir zwei Ansätze um obere Schranken für die Lösungsgröße zu berechnen, welche genutzt werden können, um die Suche im Lösungsraum früher zu beenden. Zusätzlich untersuchen wir das Ausnutzen von Symmetrien im Lösungsraum, um den Lösungsraum zu verkleinern. Wir untersuchen weiters die Anwendung eines Algorithmus zur Findung von einer größten stabilen Menge als Alternative zu bestehenden Lösungsansätzen. In unserer experimentellen Auswertung untersuchen wir diese Techniken und zeigen, dass ein heuristischer Algorithmus für stabile Mengen existierende Algorithmen sowohl in Bezug auf Laufzeit als auch in Bezug auf Ergebnisqualität überbieten kann, wenn die Laufzeit per Eingabe-Graphen limitiert wird. Weiters präsentieren und untersuchen wir eine Vorverarbeitungs-Phase für einen existierenden Algorithmus, welche für eine Gruppe an Eingabe-Graphen in der Lage ist, die Laufzeit um über 80% zu reduzieren.

Contents

1 Introduction

Graphs are used to model a wide variety of problems in various different disciplines. Given two graphs, it is often necessary to find the largest induced common subgraph between the two graphs. This is called the Maximum Common Subgraph problem and is NP-complete [GJ90, SAKZ+05]. Different variants of the Maximum Common Subgraph problem exist, in this work we focus on the Maximum Common *Induced* Subgraph problem (MCS), which requires the subgraphs to be vertex-induced on the input graphs. This problem finds applications in a variety of fields, such as chemistry and molecular science [CVM77, ER11, GFM+14, GARW93, RW02], where Maximum Common Subgraphs can aid the computational drug discovery. Additional applications can be found in the fields of pattern recognition [BFG+02, MP03] and source code analysis [DCH97]. Furthermore, Maximum Common Subgraphs can be used as a means of detecting malware [PRS13]. Park et al. [PRS13] present an approach, which derives the so called *HotPath* from a family of malware. The HotPath represents execution behavior shared by all members of the malware family. Thus, to detect new malware, Park et al. [PRS13] propose using Maximum Common Subgraphs to determine the similarities of the execution behavior of possibly malicious software to the HotPath.

Input graphs for the MCS problem can be either undirected or directed, and vertices as well as edges may be labeled. When used in the context of biology or chemistry, the instances will mostly be undirected and vertex-labeled, as graphs are used to represent proteins and molecules. For some applications it may additionally be of interest to encode the chemical bond between two atoms via edge labels. In the context of pattern recognition, the graphs are mostly undirected and they may either be labeled or unlabeled, whereas for source code analysis the graphs contain both vertex- and edge-labels, but they may either be directed or undirected. The graphs used by Park et al. [PRS13] to determine execution behavior of software are directed as well as vertex- and edge-labeled.

Most commonly, the MCS problem is solved by reducing it to a maximum clique problem. This is done by constructing an auxiliary graph, which is essentially the product of the two input graphs, where each vertex of the first input graphs is combined with each vertex of the second input graph to create the vertices of the auxiliary graph. Using this auxiliary graph, the Maximum Common Subgraph can be determined by using any maximum clique solver [Koc01, Lev73, MNPS], where the vertices of the clique represent a mapping between vertices of the input graphs. Complementary to the maximum clique problem is the minimum vertex-cover problem, thus an alternative approach is to build the complement of the auxiliary graph and then apply a minimum vertex-cover solver [AKSRL07, SAKZ+05]. An alternative approach is to define the MCS problem as an optimization problem and solve it using constraint-programming [HMR17, NS11, VV08]. More recently, the problem has been solved by applying a partitioning scheme on the input graphs [MPT17], which does not require the auxiliary graph and instead branches on possible mappings between vertices of the input graph. At each branching step, the remaining unmapped vertices are partitioned into different sets and only vertices from the same set may be mapped onto each other. This approach has recently been extended to utilize techniques from machine learning in the branch selection process [LLJH20], where reinforcement learning is used to learn which branch may yield the highest reward, based on scores assigned to vertices. The current state-of-the-art is the partitioning based approach McSplit [MPT17], which is only inferior to a maximum clique solver on edge- and vertex labeled graphs.

1.1 Contribution

We introduce reduction rules, for reducing the size of the input graphs and thus the problem size. These rules may be applied to the input graphs themselves, and are thus approach agnostic. Additionally, we propose two new methods of computing upper bounds, which may be useful in terminating the search for the Maximum Common Subgraph early, as no optimal solution may exceed the upper bound. Next, we present an initialization phase for the machine learning approach of [LLJH20], which may be used to jump start their algorithm by computing initial scores for vertices. We show, that for some instances, this initialization phase can speed up the original algorithm by a factor of 100. We additionally propose methods for reducing the search spaces of different algorithms through the exploitation of symmetries within the graphs. Finally, we propose to solve the MCS problem by computing independent sets on the complement of the auxiliary graph and we propose problem specific changes to the current state-of-the-art independent set solver. We show, that for difficult instances, a heuristic independent set solver may significantly outperform other approaches in terms of running time and result quality.

1.2 Structure of This Thesis

First, we discuss notation and necessary preliminaries in Chapter 2. We follow with a thorough literature review regarding the related work on the MCS problem, as well as related algorithms and literature in Chapter 3. Chapter 4 describes our contributions in depth. We experimentally evaluate our contributions against state-of-the-art competitors in Chapter 5. Finally, we conclude with Chapter 6 and discuss possible future work.

2 Preliminaries

In this chapter we discuss the preliminaries and the notation used throughout the remainder of this thesis.

A *graph* $G = (V(G), E(G))$ consists of the vertex set $V(G)$ and the set of undirected edges $E(G)$. We speak of a *digraph*, if the edges are directed. *Directed edges* are denoted as (u, v) and *undirected edges* as $\{u, v\}$. The *label* of a vertex $u \in V(G)$ is a mapping of $V(G) \to \mathcal{N}$ and is denoted as $\mu_G(u)$. For edges, the label of an edge is mapping of $E(G) \to \mathcal{N}$. We denote by $\lambda_G(\{u, v\})$ the label of the edge $\{u, v\} \in E(G)$ and $\lambda_G((u, v))$ denotes the label of the directed edge (u, v). We denote by $\mu_G(V(G))$ set of vertex labels existing in G.

The *neighborhood* $N(u)$ of a vertex u denotes the set of neighbors of u, i.e. $N(u) = \{v \in V(G) : \{u, v\} \in E(G)\}$ in the undirected case, or $N(u) = \{v \in V(G) : (u, v) \in E(G)\}$ in the directed case. The *closed neighborhood* is denoted as $N[u] = N(u) \cup \{u\}$. A vertex u has *degree* $\deg(u) = |N(u)|$, i.e. the degree of a vertex is the size of its neighborhood, u is called *isolated* if $\deg(u) = 0$. A graph is *complete*, if all vertex pairs of the graph are connected by an edge, a digraph is *complete*, if all vertex pairs $\{u, v\}$ of the graph are connected by an edge (u, v) as well as an edge (v, u). The *complement* $G^{\mathcal{C}}$ of a graph G contains exactly those edges missing in G which are required to form a complete graph. Given a set of vertices $\mathcal{V} \subseteq V(G)$ the *vertex induced subgraph* $G[\mathcal{V}]$ of a graph is a subgraph of G, such that $G[\mathcal{V}] = (\mathcal{V}, \{\{u, v\} \in E(G) : u, v \in \mathcal{V}\})$

Definition 2.1. *An* isomorphism *between two graphs G_1 and G_2 is a bijective function ϕ, such that $\forall u, v : \{u, v\} \in E(G_1) \Leftrightarrow \{\phi(u), \phi(v)\} \in E(G_2)$, i.e. a bijective mapping from vertices of one graph to the vertices of the other graph, such that edges are preserved.*

Definition 2.2. *An* automorphism *is an isomorphism from a graph G to itself.*

With $i \in [x..y]$ we denote that i can take the values $x \leq i \leq y$ with $i \in \mathbb{Z}$. An *undirected path* in G is a sequence of vertices $u_1 \to u_2 \to \cdots u_i$, such that $\forall j \in [1..i-1] : \{u_j, u_{j+1}\} \in E(G)$ and no vertex appears more than once. A graph G is *connected* if there is a path between every pair of vertices. A digraph G is *weakly connected* if the underlying graph is connected.

Definition 2.3. *The Maximum Common Induced Subgraph (MCS) problem is, given two input graphs G_1 and G_2, find the largest sets of vertices $\mathcal{V}_1 \subseteq V(G_1)$ and $\mathcal{V}_2 \subseteq V(G_2)$ with $|\mathcal{V}_1| = |\mathcal{V}_2|$, such that the induced subgraphs $G_1[\mathcal{V}_1]$ and $G_2[\mathcal{V}_2]$ are isomorphic.*

Other variants of the problem exist, e.g. the Maximum Common *Connected* Subgraph (MCCS) problem, where we search for the maximum common induced subgraph, that is connected. The input graphs for the MCS problem can be directed or undirected and edges and vertices may optionally have labels, if no labels are given, we assume the unlabeled vertices and edges to all have identical labels. The labels of the input graphs have to be taken into account, such that only vertices and edges of matching labels may be mapped onto each other.

Definition 2.4. *A* clique *is a set of vertices $C \subseteq V(G)$, such that $G[C]$ induces a complete graph, i.e. all vertices in C are adjacent to each other.*

A clique C is *maximal* if no further vertices can be added to it without violating the clique constraint. A *maximum clique* is a clique of a given graph, such that no larger clique exists. An *independent set* is a set of vertices of $\mathcal{I} \subseteq V(G)$, such that $\forall u, v \in \mathcal{I} : u \notin N(v)$, meaning that the vertices of $\mathcal{I}$ are not adjacent. As with the clique, a *maximal* independent set is an independent set such that no further vertices can be added without violating the independent set constraint, and a

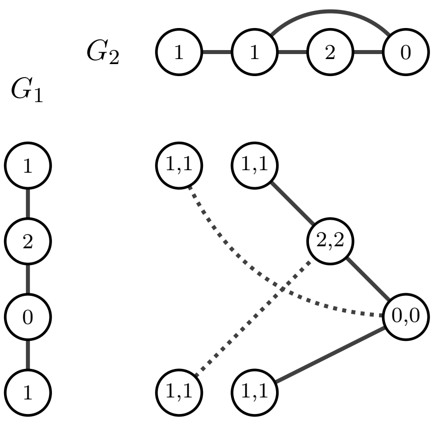


Figure 2.1: Example of two input graphs with vertex labels and the resulting product graph. The edge-style in the product graph encodes how an edge was created: solid edges are created through the existence of edges in the input graphs whereas dashed edges are created through the absence of edges between vertices of the input graphs.

maximum independent set is a largest of those sets. Note the relationship between cliques and independent sets: a maximum clique in G is a maximum independent set in $G^{\mathcal{C}}$ and a maximum independent set in G is a maximum clique in $G^{\mathcal{C}}$. Additionally note, that neither the maximum independent set nor the maximum clique are necessarily unique. A *vertex-cover* is a set of vertices $\mathcal{V} \in V(G)$, such that $V(G) \setminus \mathcal{V}$ is an independent set. The vertex-cover problem is to find a minimum number of vertices, such that at least one endpoint of each edge is in the vertex-cover. The problem of finding a maximum independent set is thus equivalent to finding a minimum vertex-cover. A complete graph with n vertices has a minimum vertex-cover of size $n-1$.

2.1 Product Graph

The product graph (or *association graph*, *modular product of graphs* or *compatibility graph*) PG of two graphs G_1 and G_2, subject to labels is defined as follows:

Definition 2.5. $V(PG) = \{r = uv \in V(G_1) \times V(G_2) : \mu_{G_1}(u) = \mu_{G_2}(v)\}\}$

Definition 2.6. *For $r = uv, s = u'v' \in V(PG)$ with $u, u' \in V(G_1)$ and $v, v' \in V(G_2)$, $\{r, s\} \in E(PG)$ if and only if either*

1. $\{u, u'\} \in E(G_1) \wedge \{v, v'\} \in E(G_2) \wedge \lambda_{G_1}(\{u, u'\}) = \lambda_{G_2}(\{v, v'\})$ *or*
2. $\{u, u'\} \notin E(G_1) \wedge \{v, v'\} \notin E(G_2)$

Definition 2.7. *For directed input graphs, the product graph is constructed analogous to the undirected case.*

Note, that the product graph is completely unlabeled and consists of undirected edges, regardless of the input graphs. An example of two input graphs and the resulting product graph is given in Figure 2.1. To compute the MCS we could now apply a maximum clique algorithm on the product graph, where any vertex of the clique represents the mapping of two vertices from the input graphs [Lev73]. However, we can instead compute the product graph complement and compute either a maximum independent set or a minimum vertex-cover to arrive at an optimal solution. The complement $PG^{\mathcal{C}}$ of the product graph can be computed directly by employing the following rules:

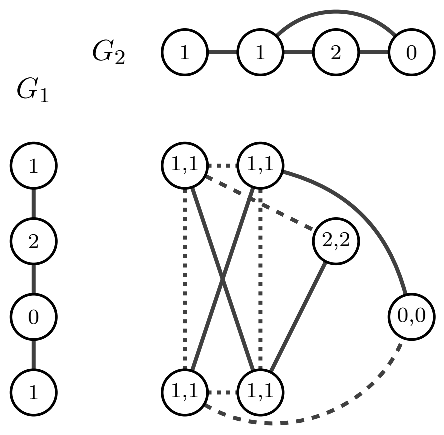


Figure 2.2: Example of two input graphs with vertex labels and the resulting product graph complement. Solid edges of the product graph correspond to edges created with ${G_1}^{\mathcal{C}} \times G_2$, dashed edges correspond to edges created with $G_1 \times {G_2}^{\mathcal{C}}$ and dotted edges correspond row and column clique edges.

Definition 2.8. *$V(PG^{\mathcal{C}}) = \{r = uv \in V(G_1) \times V(G_2) : \mu_{G_1}(u) = \mu_{G_2}(v)\}\}$*

Definition 2.9. *For $r = uv, s = u'v' \in V(PG)$ with $u, u' \in V(G_1)$ and $v, v' \in V(G_2)$, $\{r, s\} \in E(PG)$ if and only if either*

1. *$\{u, u'\} \in E({G_1}^{\mathcal{C}}) \wedge \{v, v'\} \in E(G_2)$ (denoted as ${G_1}^{\mathcal{C}} \times G_2$) or*
2. *$\{u, u'\} \in E(G_1) \wedge \{v, v'\} \in E({G_2}^{\mathcal{C}})$ (denoted as $G_1 \times {G_2}^{\mathcal{C}}$) or*
3. *$\{u, u'\} \in E(G_1) \wedge \{v, v'\} \in E(G_2) \wedge \lambda_{G_1}(\{u, u'\}) \neq \lambda_{G_2}(\{v, v'\})$*

Definition 2.10. *For directed input graphs, the product graph complement is constructed analogous to the undirected case.*

As with the product graph, the product graph complement is completely unlabeled and edges are undirected. Figure 2.2 gives a small example of two input graphs and the resulting product graph complement. Note the construction of the product graph complement – the rows contain all vertices created by fixing one vertex in G_1 and combining that vertex with all vertices of identical label in G_2, whereas the columns consist of all vertices created by fixing one vertex in G_2 and combining that with vertices of identical label in G_1. This notion of *rows* and *columns* in the product graph (complement) will be relevant later on. We expect the complement to be denser than the product graph itself in most cases. Let $n_{G_1} = |V(G_1)|$ and $m_{G_1} = |E(G_1)|$, analogously n_{G_2} and m_{G_2} for G_2. If we consider undirected and unlabeled input graphs, the product graph PG will consist of $n_{PG} = n_{G_1} n_{G_2}$ vertices and

$$m_{PG} = 2(m_{G_1} m_{G_2} + (\frac{(n_{G_1} - 1)n_{G_1}}{2} - m_{G_1})(\frac{(n_{G_2} - 1)n_{G_2}}{2} - m_{G_2}))$$

edges. Due to the construction of the product graph we first introduce edges for all combination of edges of the two input graphs. Next, we pair all *missing* edges from both input graphs, to construct edges in the product graph. Finally, given two vertices $u, u' \in V(G_1)$ and two vertices $v, v' \in V(G_2)$ we arrive at four different vertices in the product graph. Thus combining the edges $\{u, u'\} \in V(G_1)$ and $\{v, v'\} \in V(G_2)$ yields two unique edges in $E(PG)$, namely $\{uv, u'v'\}$ and $\{uv', u'v\}$. Therefore, we multiply the number of edges by two, to arrive at the final edge count of the product graph. Conversely, the product graph complement $PG^{\mathcal{C}}$ will have

$$m_{PG^{\mathcal{C}}} = \frac{(n_{PG} - 1)n_{PG}}{2} - m_{PG}$$

edges, i.e. exactly those edges, which are missing in $E(PG)$ to form a complete graph. The product graph complement is sparse when either both input graphs are very dense, or both input graphs are very sparse. An evaluation of the densities of our product graph complement graphs can be found in Section 5.3.

3 Related Work

In this chapter we discuss prior work done on the MCS problem as well as some closely related problems. We first discuss various approaches to the MCS problem. First we discuss prior work using clique-based or vertex-cover based approaches to the MCS problem. Second, we review variants utilizing Constraint Satisfaction Programming as well as a partitioning based approach and its extension utilizing machine learning. Finally, we describe an algorithm used for graph isomorphism testing, a tool for computing automorphisms, as well as a maximum independent set solver used in our work.

3.1 Maximum Common Subgraphs

We first discuss the prior work done on the MCS problem, introducing various different approaches as well as some important results and we present the current state-of-the-art.

3.1.1 Clique-based Approaches

Traditionally, the MCS problem has been solved by building the product graph and computing the maximum clique on that graph. This variant of solving the MCS problem was first described by Levi [Lev73]. Here we describe the state-of-the-art in clique solvers, as well as some other notable work.

Based on the Bron-Kerbosch algorithm [BK73], which enumerates all maximal cliques of a given graph, Koch [Koc01] presents different new variants, based on properties of the product graph. The Bron-Kerbosch algorithm [BK73] operates on three sets C, P and S, where C stores the current clique, P contains the set of vertices, which may still enlarge the current clique and S contains a set of vertices, which may not be added to C anymore, as the vertices in S have already been processed. In every branching step, a vertex $u \in P$ is added to C and P is set to be $P \cap N(u)$ and S is set to $S \cap N(u)$. With these updated values, the algorithm calls itself recursively. After the recursive branching returns, S is enlarged by u, indicating that no further unique maximal clique which include u can be found.

Koch [Koc01] extends this algorithm to only find cliques representing Maximum Common *Connected* Subgraphs, and notes, that for the non-connected MCS problem many automorphisms may occur, which lead to many equivalent solutions, and therefore a large search tree. Even though Koch [Koc01] makes this observation, to the best of our knowledge, automorphisms have not yet been explicitly exploited for the MCS problem in the literature.

McCreesh et al. [MNPS] note that the downside of the approach of Koch lies in the base-algorithm used, as the Bron-Kerbosch algorithm is a maximal clique *enumeration* algorithm, thus potentially very inefficient at finding the *maximum* clique.

To the best of our knowledge, the state-of-the-art in maximum clique-solvers is due to Li et al. [LJM17] and is called MoMC. Most modern Branch-And-Bound clique-solvers partition the search space in the following way: at each branching step – given a lower bound r, the vertices are partitioned into a set A, such that the vertices of A may form at most a clique of size r, and a set B of branching vertices. Branching only has to be performed on vertices of the set B, therefore it is beneficial to have B be as small as possible. MoMC combines two different partitioning approaches, one based on a reduction to MaxSAT, the other based on a static vertex ordering: Given a vertex-ordering O, all vertices of A should be larger than the vertices of B w.r.t. the ordering O. At each branching step, MoMC computes a partition using the MaxSAT reduction, if that yields an empty set for B, no branching has to occur at this step. If B is not empty, an alternate B_s is built by adding all vertices v to B_s, such that v is smaller than the largest vertex of B w.r.t. O. Depending on a parameter α and the ratio of sizes between B and B_s it is decided, which of the two sets is chosen. If $\frac{|B|}{|B_s|} < \alpha$, then B is chosen, otherwise B_s is chosen. Li et al. show that given $\alpha = 0.6$ this approach outperforms previous algorithms significantly, and it has a slight advantage over similar implementations, which only utilize either one of the partitioning methods. Note, that a single-partitioning-approach implementation can be achieved by setting α to 0 (Vertex-order partition) or 1 (MaxSAT).

3.1.2 Vertex Cover-based Approaches

As discussed in Chapter 2, the MCS problem can be solved by building the product graph complement, which may then be used to compute a minimum vertex-cover in order to arrive at an optimal result.
Suters et al. [SAKZ$^+$05] present such an approach, which applies the branch-and-bound approach to the vertex-cover problem. For their algorithm, they make use of the construction of the product graph complement, which introduces row- and column cliques for each vertex. Let $r = uv, u \in G_1, v \in G_2$ be any vertex in $PG^{\mathcal{C}}$, r belongs to a row-clique which corresponds to u being mapped to any applicable vertex in G_2, and to a column-clique, which corresponds to v being mapped to any applicable vertex in G_1 (see Chapter 2 for details). Any minimum vertex-cover has to either include all of any row (column)-clique or exclude one single vertex of the clique. Therefore, for each vertex r it is required to branch into one branch which excludes r and instead includes the remainder of its cliques, and one branch which includes r and all its clique neighbors. Let k denote the size of a clique, there are $k + 1$ possible branches for each clique, k branches where one vertex each is excluded, plus one additional branch, where no vertex is excluded. Let $n_1 = |V(G_1)|$ and $n_2 = |V(G_2)|$, the algorithm proposed by Suters et al. [SAKZ$^+$05] iterates over the n_1 rows of the product graph complement and in each iteration branches up to $n_2 + 1$ times, as discussed before. Therefore, this algorithm has a worst-case running time complexity of $\mathcal{O}((n_2 + 1)^{n_1})$. Note, that this bound is not tight, as in deeper branching levels the number of possible branches may be smaller than $n_2 + 1$, as an excluded vertex immediately implies all its clique neighbors are part of the vertex-cover and can therefore not be branched upon.
Another approach based on vertex-cover, which does not utilize either the product graph or its complement at all, is proposed by Abu-Khazam et al. [AKSRL07]. For their algorithm, Abu-Khazam et al. compute a vertex-cover C of G_1 and try to match the vertices of the cover to vertices of G_2 by utilizing an exhaustive search. Thus, they arrive at a number of candidate subgraphs of $G_1[C]$, which may be extended by further vertex mappings to achieve a maximum common subgraph. This extension of candidate sets to maximal common subgraphs is done by computing a maximum independent set in $G_1 \setminus C$. Let $n_2 = |V(G_2)|$, the overall running time complexity, dependent on $k = |C|$, which denotes the the size of the vertex cover of G_1, is $\mathcal{O}(3^{n_2/3}(n_2 + 1)^k)$.
To the best of our knowledge, neither of these two vertex-cover based approaches has been evaluated experimentally in the literature.

3.1.3 Constraint Programming-based Approaches

The MCS has been defined as a *constraint satisfaction problem* multiple times throughout the literature [HMR17, MNPS, NS11, VV08]. Constraint satisfaction problems (CSP) are defined by a set of decision variables X, their possible domain of values D as well as a set of constraints C. This section explores some of the most recent approaches found.
Vismara and Valery [VV08] present a simple CSP formulation for the MCS problem, where each vertex $u \in V(G_1)$ has one decision variable x_u, the domain of each variable is $V(G_2) \cup \perp$, where $\perp$ denotes, that a vertex is not matched, otherwise a vertex u of G_1 is matched to a vertex of G_2

as indicated by the decision variable x_u. A simple constraint is required, which enforces, that for an edge $\{u, v\} \in G_1$ either x_u or x_v is set to $\perp$ or $\{x_u, x_v\} \in E(G_2)$. Finally, as the mapping has to be bijective, a uniqueness-constraint is required, such that each $x_u \in X \neq \perp$ has a unique value. To achieve this, Vismara and Valery [VV08] define a set of binary constraints between each pair of possible matchings of vertices. This CSP approach can be extended to graphs with labeled vertices, by simply modifying the domain of each variable x_u to only those vertices of G_2 which have the same label as u.

Ndiaye and Solnon [NS11] build on the work of Vismara and Valery [VV08] by replacing the binary constraint between all pairs of possible vertex matchings by a soft constraint, which maximizes the number of unique assignments (except for assignments to $\perp$) whilst maintaining the bijectivity constraint.

For a long time it was believed that constraint programming approaches are the fastest method of solving the MCS problem [MNPS], however, in their experiments, McCreesh et al. [MNPS] showed, that state-of-the-art clique solvers are superior on graphs with labeled vertices and competitive on unlabeled graphs.

Hoffmann et al. [HMR17] present a constraint-programming algorithm for a slightly different problem: given two graphs G_1 and G_2 and a natural number k, check if all but k vertices of G_1 can be mapped to G_2. This modified problem lies between *subgraph isomorphism*, which asks to find a pattern graph in a target graph, and the MCS problem. To solve this modified problem, Hoffmann et al. [HMR17] slightly modify the constraint-programming approach by Ndiaye and Solnon [NS11]. First, Hoffmann et al. alter the approach to be bit-parallel by storing domain values as bit sets and graphs as adjacency matrices. Additionally, instead of one $\perp$ value to denote a non-matched vertex, Hoffmann et al. introduce k different $\perp$ values. Finally, this algorithm can be turned into a MCS solver, by setting $k = 0$ and iteratively increasing k until a solution is found. This algorithm is referred to as $k\downarrow$ and differs to other approaches in that it attempts to find solutions of *decreasing* size, whereas most solvers try to find improved larger solutions until no larger solution can be found. According to their experiments, which were performed only on unlabeled instances, $k\downarrow$ outperforms other constraint-programming approaches, as well as state-of-the-art clique solvers.

3.1.4 Partitioning-based Approaches

McCreesh et al. [MPT17] present a partitioning approach, which forgoes computing the product graph or its complement completely, and instead, their algorithm McSplit, purely operates on the two input graphs. To compute a maximum common subgraph solution, McSplit branches on mappings between vertices of the two input graphs, computing new labels for all vertices in the input graphs to quickly determine, which mappings may be added to the current solution. Each vertex of the two input graphs start off with empty labels. When two vertices $u \in G_1$ and $v \in G_2$ are mapped onto each other, in the input graphs the neighbors of $N(u)$ and the neighbors of $N(v)$ have a 1 appended to their label, indicating that these vertices are adjacent to the mapped vertex, whereas non-adjacent vertices in the input graphs have a 0 appended to their label. This update to the labels leads to a partitioning of the vertices, such a partition with corresponding label is referred to as a *domain* by McCreesh et al. Next, only vertices of the same domain may be mapped onto each other, otherwise the maximum common induced subgraph property would be violated. See Figure 3.1 and Table 3.1 for a small example, of how McSplit operates.

This labeling not only filters which vertices may be mapped onto each other, it additionally can be used to compute an upper bound during the branching. Let $D_i^{G_1}$ and $D_i^{G_2}$ denote the set of vertices in domain i for G_1 and G_2 respectively. The upper bound is simply computed by $ub = \sum_{\forall i} min(|D_i^{G_1}|, |D_i^{G_2}|)$. If for any branch this upper bound is less than or equal to an already computed lower bound, the search at that branch can be terminated, as no improvement is possible.

McSplit can be extended to allow for vertex- and edge labels, as well as for directed graphs and connected subgraphs. To resolve ambiguities in the selection of branches, if multiple equivalent possibilities exist, McCreesh et al. [MPT17] propose to branch on nodes with large degree first. The authors note, that this heuristic was chosen based on the result of some small scale experiments, however, different approaches may be applied as well, further research into that topic is left for future work.

In their experiments the authors demonstrate that McSplit outperforms constraint-programming approaches on all graph classes, and state-of-the-art clique solvers on unlabeled graphs, whereas clique solvers still perform best on labeled graphs.

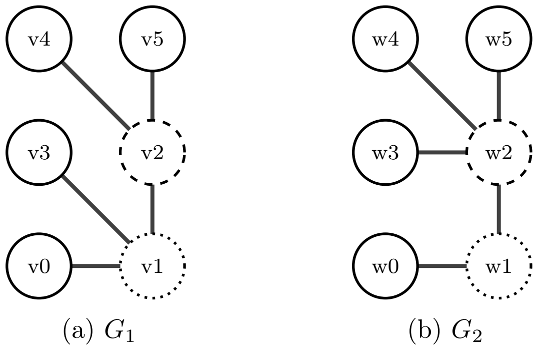


Figure 3.1: Two input graphs for the McSplit example. Vertex labels are encoded as the style of the outlines of vertices. McSplit produces the following mapping: $v0 \to w0$, $v1 \to w1$, $v2 \to w2$, $v4 \to w3$ and $v5 \to w4$.

3.1.5 Machine Learning-based Approaches

As machine learning finds more wide-spread uses, it is not surprising, that machine learning techniques are applied to graph theoretical problems as well. In this section, we discuss a recent approach, which employs techniques from machine learning to the MCS problem.

Built on top of McSplit [MPT17], Liu et al. [LLJH20] propose to use reinforcement learning as the heuristic for branch-selection. The authors note, that their approach is not confined to McSplit and could be applied to other solvers as well, however, at the time of publishing of their work, McSplit constituted the state-of-the-art, therefore the technique was applied to that solver.

In reinforcement learning, agents need to make decisions based on an expected numerical gain value. After a decision has been made, the actual gain is observed and turned into a reward for the agent, the goal for the agent is to maximize the reward by taking actions with large gains. In the context of the McSplit algorithm, the algorithm is perceived as an agent, which has the goal of reaching a leaf branch early on, as to reduce the size of the branching tree by achieving a good lower bound. At each branch, each choice of possible mappings are considered as different actions the agent can take, each action with its own reward value. For the computation of the expected reward value, Liu et al. [LLJH20] propose a method, which computes the rate of reduction of the upper bound value, which bounds the solution size possible in the current branch. Once the upper bound of a branch is less than or equal to the currently largest solution, no improvement can be made in a branch and thus the branch can be terminated. Each vertex has a cumulative score value of rewards and branches are chosen based on this score, such that vertices with large score are branched on first. Note, that these scores are learned from scratch for each instantiation of the algorithm with a pair of input graphs. Liu et al. [LLJH20] show that their approach (named McSplit+RL) improves on McSplit consistently throughout all experiments. However, their experiments simply compare McSplit+RL with McSplit, but not with any other approach.

G_1 Vertex	G_1 Label	G_2 Vertex	G_2 Label
$v0$	0	$w0$	0
$v1$	1	$w1$	1
$v2$	2	$w2$	2
$v3$	0	$w3$	0
$v4$	0	$w4$	0
$v5$	0	$w5$	0

(a) Initial labels, $ub = 6$

G_1 Vertex	G_1 Label	G_2 Vertex	G_2 Label
$v0$	01	$w0$	01
$v2$	21	$w2$	21
$v3$	01	$w3$	00
$v4$	00	$w4$	00
$v5$	00	$w5$	00

(b) After mapping $v1$ to $w1$, $ub = 5$

G_1 Vertex	G_1 Label	G_2 Vertex	G_2 Label
$v0$	010	$w0$	010
$v3$	010	$w3$	001
$v4$	001	$w4$	001
$v5$	001	$w5$	001

(c) After mapping $v2$ to $w2$, $ub = 5$

G_1 Vertex	G_1 Label	G_2 Vertex	G_2 Label
$v3$	0100	$w3$	0010
$v4$	0010	$w4$	0010
$v5$	0010	$w5$	0010

(d) After mapping $v0$ to $w0$, $ub = 5$

G_1 Vertex	G_1 Label	G_2 Vertex	G_2 Label
$v3$	01000	$w4$	00100
$v5$	00100	$w5$	00100

(e) After mapping $v4$ to $w3$, $ub = 5$

G_1 Vertex	G_1 Label	G_2 Vertex	G_2 Label
$v3$	010000	$w5$	001000

(f) After mapping $v5$ to $w4$, $ub = 5$

Table 3.1: Example of McSplit for Figure 3.1. The different labels are used after each step to determine an upper bound. The upper bound ub is given at each step. The final solution ends up as $v0 \to w0$, $v1 \to w1$, $v2 \to w2$, $v4 \to w3$ and $v5 \to w4$.

3.2 Automorphisms and Isomorphisms

Weisfeiler and Leman introduce an algorithm, which is known as the *Weisfeiler-Leman algorithm* and it is useful for testing for graph isomorphisms [WL68]. Additionally, the Weisfeiler-Leman algorithm has recently been of interest in the field of machine learning by first computing *node embeddings* for each vertex of the graph, which are then used as the feature vector for nodes [MFK21, SSvL$^+$11, TGL$^+$19]. These node embeddings can be computed using i iterations of the Weisfeiler-Leman algorithm, where for each vertex v a vector x_v is computed, containing the label the vertex obtained in each of the i iterations of Weisfeiler-Leman.

The algorithm works as follows

1. *Initialize*: assign each vertex their initial label (0 if no labels are given)
2. *Refine*: on each vertex aggregate the labels of its closed neighborhood into a multi set, map that multi set onto a new label and assign it to the vertex
3. *Repeat* the second step until the labeling is stable, i.e. the relative relationship between all vertices regarding their label remains unchanged

If for two graphs G_1 and G_2 the stable labelings are non-isomorphic then the graphs themselves have to be non-isomorphic as well. Note, that isomorphic labelings are a necessary but not sufficient requirement for graph isomorphism. A simple example, where the Weisfeiler-Leman

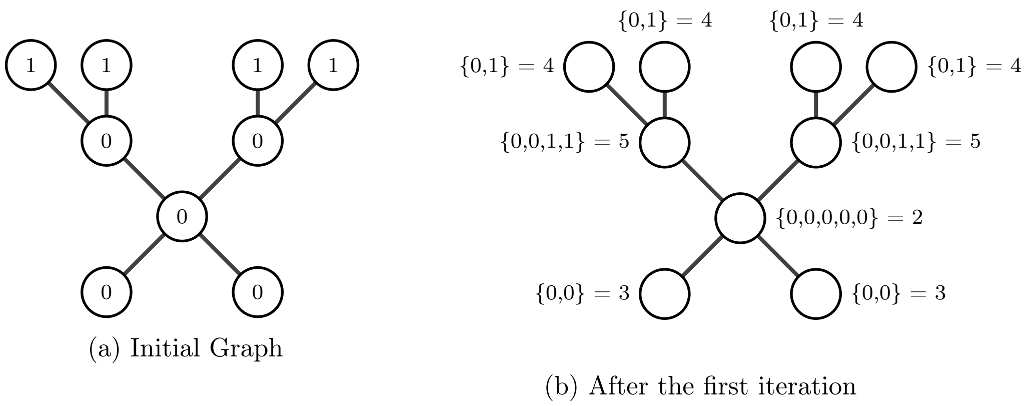


Figure 3.2: One iteration of the Weisfeiler-Leman algorithm on a small graph

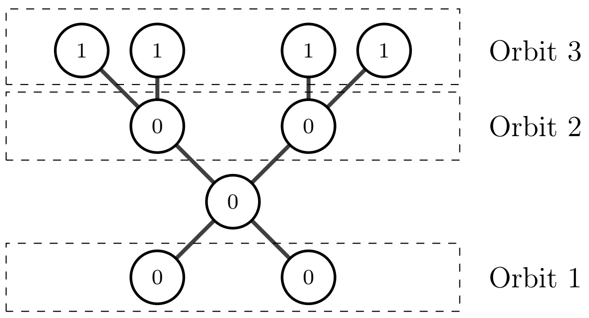


Figure 3.3: Example of orbits in a graph. Vertices are labeled, non-trivial orbits are represented by the dashed rectangles.

algorithm fails to show non-isomorphism is one circular graph consisting of 6 vertices and one graph consisting of two separate triangles, all vertices have the same label. The algorithm arrives at the same stable labeling for both graphs, whereas the graphs are clearly not isomorphic to each other. Therefore, the Weisfeiler-Leman algorithm is useful for determining non-isomorphism, to prove graph isomorphisms further computations are required, which are not relevant here and are thus not discussed.

The new vertex labels after each iteration of the algorithm can be used to determine structural information about a graph. After the n-th iteration, vertices which have the same label all have equivalent n-hop neighborhoods w.r.t. vertex labels. Figure 3.2 shows a small example with one iteration of the algorithm executed. For each vertex u we have the multi set of the labels of $N[u]$ after the first iteration, and the new label to which that multi set is mapped. We can clearly observe how vertices with the same 1-hop neighborhood and same initial label end up with the same new label after the first iteration. Note, that the labeling is already stable after the first iteration, therefore no further iterations are given.

Nauty is a tool for determining graph *automorphisms* by McKay et al. [MP14]. Given a (labeled) graph G, Nauty computes all automorphisms of the graph, as well as so called *orbits*. An *orbit* O is a maximal set of vertices, such that $\forall u, v \in O$: there exists an automorphism ϕ in G, such that $\phi(u) = v$. An orbit can thus be considered an equivalence class of vertices. See Figure 3.3 for a small example of non-trivial orbits in a graph.

3.3 Independent Set

As mentioned in Chapter 2, by building the product graph complement, the MCS can be determined using either a vertex-cover or an independent set solver. We consider an independent set solver due to Lamm et al. [LSS$^+$19] called KaMIS, which has been shown to perform well in practice. Although KaMIS targets the *weighted* independent set problem, i.e. the problem of finding an independent set of maximum weight, the same algorithm can be applied to the unweighted problem as well. Their algorithm combines a branch-and-bound approach with reduction rules, which decrease the size of the input graph. Together, this approach is referred to as *Branch-And-Reduce.* These reductions consist of rules which can add vertices to the solution immediately, remove some vertices from the graph entirely or fold some vertices into a single super vertex, such that at the end of the algorithm this super vertex is unfolded and the solution extended [LSS$^+$19]. We next give a brief overview of a few reduction rules. The *neighborhood* reduction puts a vertex $u \in V(G)$ into solution, if the weight of u is larger than the weight of $u's$ neighborhood. Given two vertices $u, v \in V(G)$, u is said to *dominate* v if $N[u] \supseteq N[v]$. In such a case, v may always replace u in an independent set, and thus u can be removed from the graph. Two vertices $u, v \in V(G)$ are *twins*, if their neighborhood $N(u) = N(v) = \{p, q, r\}$. In this case, u and v can be contracted into a single vertex $\{u, v\}$ with weight equal to the sum of the weights of u and v, and at a later time it is decided whether u and v are in the solution or any subset of their neighbors are. If there exists a clique in G, such that at least one vertex u of the clique has no non-clique neighbors, u can be put into solution. Given a vertex $u \in V(G)$, if u has neighborhood $N(u) = \{p, q\}$, such that p and q are not adjacent to each other, the three vertices $\{u, p, q\}$ are *folded* into a single vertex with the combined weight of p and q.

When the algorithm is first called, KaMIS initially applies the reduction rules and thus attempts to reduce the size of the graph right away. Next, the Branch-And-Reduce algorithm searches for an independent set in each graph component separately. After each branching step where a new vertex is added to the solution, the reduction rules are called again. The idea is, that adding a vertex v to solution - and thus excluding all vertices $u \in N(v)$ - may change the graph such that the reduction rules may apply again and thus simplify the problem. Additionally, adding a vertex v to solution may split the remaining graph into multiple components. If that is the case, KaMIS will again apply the algorithm on each component separately.

In addition to the exact solver, Lamm et al. [LSS$^+$19] present a heuristic Local Search algorithm as well. This Local Search algorithm utilizes the reductions for an initial reduction as well, however, during the search the reductions are not used again. Note, that the Local Search algorithm can be used to obtain an initial lower bound for the Branch-And-Reduce algorithm. The Local Search starts off by applying the data reduction rules to reduce the size of the input graph. On this reduced graph, the Local Search finds some initial solution using a greedy heuristic. Next, this solution is modified in k iterations, by either performing a combination of so called *1-2 swaps* or by forcing vertices into the solution. During 1-2 swaps KaMIS attempts to find a vertex u currently in solution, such that u can be removed and two neighbors of u can be added to the solution instead. Forcing a vertex u into the solution requires removing all neighbors of u from solution, if any currently are in solution. After these steps the solution may not be *maximal*, i.e. further vertices may be added to solution. This process is repeated for some fixed number of iterations.

4 Engineering MCS Algorithms

In this chapter we present our contributions, starting with methods of computing upper bounds on the product graph complement. Next, we introduce a method for mapping vertices before any solver is applied, followed by a rule which may be applied to exclude vertices from the mapping altogether. We follow with a description of how automorphisms may be exploited to reduce the search space, followed by two different methods of how reinforcement learning approaches may be enhanced by computing initial scores based on the neighborhoods of vertices. Finally, we discuss some MCS specific changes to an independent set solver. An overview of our suggested improvements and to which algorithm they may apply can be found in Table 4.1

4.1 Computing Better Upper Bounds

Upper bounds are useful in determining when a solution to a problem cannot be improved upon any further. That is, given an upper bound ub for a MCS instance, no solution may be larger than ub. Therefore it is desirable to compute upper bounds, as closely as possible to the size of the optimal solution. Let $L_l^{G_1}$ and $L_l^{G_2}$ denote the number of vertices with label l in G_1 and G_2 respectively. A trivial upper bound is achieved by the following equation (see Section 3.1.4):

$$ub = \sum_{\forall l} \min\left(L_l^{G_1}, L_l^{G_2}\right) \tag{4.1}$$

Here, we propose two new methods for computing upper bounds on the product graph complement. Both methods center around the idea of having two columns (rows) of the product graph complement be a clique, i.e. the vertices of the two columns (rows) induce a complete graph, which indicates that only one vertex of these two columns (rows) may be in the solution. Unless explicitly specified, we will now always only speak of columns, though everything applies to rows as well.

Lemma 4.1.1. *Let $X_l^{G_1}$ and $X_l^{G_2}$ be initialized to $L_l^{G_1}$ and $L_l^{G_2}$, respectively, for all l. Given the columns of the product graph complement which represent mappings to the vertices $u, v \in G_2$ with $\mu_{G_2}(u) = \mu_{G_2}(v)$, if the two columns form a clique, $X^{G_2}_{\mu_{G_2}(u)}$ can be decremented by one and thus potentially the upper bound reduced. Once a column has been used to decrement the label, it cannot be used again. This holds analogously for G_1 and the rows of the product graph complement.*

Proof. Let $C(u)$ denote the column in which a vertex u is located. Given that the vertices of two columns $C(u)$ and $C(v)$ with label l are a clique, adding any one vertex w of these columns to the independent set immediately excludes all $s \in N(w)$ and thus all other vertices of the columns $C(u)$ and $C(v)$. Therefore, either any vertex in $C(u)$ or any vertex in $C(v)$ may be in the solution, but not vertices from both columns. Hence we can decrease the number of vertices with label l which can be mapped. Note, that this is restricted to a single pair of columns. Assume columns $C(u)$, $C(v)$ and $C(w)$ all with label l, if $C(u)$ and $C(v)$ as well as $C(u)$ and $C(w)$ are cliques, but $C(v)$ and $C(w)$ are not, the counter for label l may only be decremented by one, as at the same time vertices of columns $C(v)$ and $C(w)$ may be in the solution. Therefore, once a column has been used to decrement a counter, it cannot be used again in this approach. This holds analogously for rows. □

After applying this method to all possible pairs of columns with matching labels, Equation 4.1 for computing the upper bound can be used to re-compute the upper bound, potentially yielding a smaller bound. This algorithm is shown in Algorithm 1.

Algorithm 1: Pseudo-Code depicting the improvement of the upper bound computation

```
Data: $G_1, G_2, PG^{\mathcal{C}}$
Result: updated upper bound $ub$
$n \leftarrow |V(G_2)|$
// $n \times n$ matrix of edge counters between columns
$M \leftarrow (\cdots)$
forall $\{u, v\} \in E(PG^{\mathcal{C}})$ do
    // Assume column() returns the column in which a vertex is located
    $i \leftarrow$ column($u$)
    $j \leftarrow$ column($v$)
    $M_{i,j} \leftarrow M_{i,j} + 1$
    $M_{j,i} \leftarrow M_{j,i} + 1$
end
$S \leftarrow \emptyset$
for $i \leftarrow 1$ to $n$ do
    if $i \in S$ then
        continue
    end
    for $j \leftarrow i + 1$ to $n : j \notin S$ do
        if $M_{i,j} = |col_i| * |col_j| \wedge \mu_{G_2}(i) = \mu_{G_2}(j)$ then
            // Columns $i$ and $j$ form a clique
            $X^{G_2}_{\mu_{G_2}(i)} \leftarrow X^{G_2}_{\mu_{G_2}(i)} - 1$
            $S \leftarrow S \cup \{j\}$
            break
        end
    end
end
$ub = \sum_{\forall l} \min(X^{G_1}_l, X^{G_2}_l)$
return $ub$
```

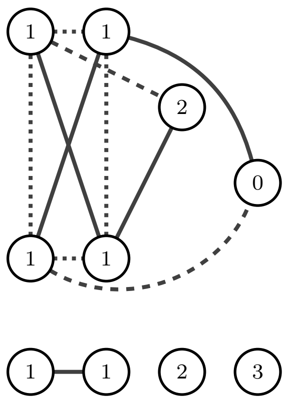


Figure 4.1: Example product graph complement with the column-contracted graph used for the upper bound computation. The vertices are labeled with $l \in \{1, 2, 3\}$.

As an example for this method, consider the product graph complement introduced in Chapter 2 and depicted in Figure 2.2: Using the formula above, we get an upper bound of $ub = 4$. However, the columns corresponding to vertices of G_2 with label 1 induce a complete graph. Thus by applying Lemma 4.1.1, we can reduce $L_1^{G_2}$ from 2 to 1 and by that reduce the upper bound to $ub = 3$.

This approach is aided by the fact, that all columns of the product graph complement form cliques, therefore it suffices to check whether between two columns all vertices are fully connected. To compute these cliques, we can iterate over all edges of the graph and keep counters for the number of edges that connect one column to another column. Let cu denote the number of vertices, which are located in column $C(u)$. If for two columns $C(u)$ and $C(v)$ there are $|cu| * |cv|$ edges between the columns, the two columns form a clique. Therefore, we then iterate over all pairs of columns with matching labels and check this condition. Overall, with $m = |E(PG^{\mathcal{C}})|$ and $n_{G_2} = |V(G_2)|$ we arrive at a running time complexity of $\mathcal{O}(m + n_{G_2}^2)$. Analogously for the rows of the product graph complement we simply replace n_{G_2} with $n_{G_1} = |V(G_1)|$.

As mentioned before, this method of reducing the upper bound is dependent on two columns having the same label. However, not all graphs in the context of the MCS problem are labeled. Additionally, two columns with different labels may form a clique as well, though that case is irrelevant for the previous approach. Therefore, we propose an additional approach, which - although being more costly - has the potential to alleviate both downsides.

To achieve this, we first create a new graph with $n = |V(G_2)|$ ($n = |V(G_1)|$) vertices, i.e. by contracting the columns (rows) of the product graph complement, such that there is an edge between two vertices if the two corresponding columns (rows) form a clique. We call this the *column contracted graph* G_{col} (*row contracted graph* G_{row}). We then arrive at an upper bound by computing the independent set of this contracted graph.

Lemma 4.1.2. *The size of a maximum independent set on a column contracted graph (row contracted graph) is an upper bound to the size of the MCS on the product graph complement.*

Proof. There is an edge between two vertices u and v in G_{col} if and only if the corresponding columns $C(u)$ and $C(v)$ form a clique. Suppose for a given product graph complement *opt* denotes the optimal solution to the MCS problem. Therefore, there have to be *opt* columns in $PG^{\mathcal{C}}$, from which vertices can be chosen, such that the vertices are independent. This implies that there are at least *opt* columns, which do not form cliques with each other, and by construction of G_{col} therefore at least *opt* vertices, which are independent of each other. The maximum independent set of G_{col} therefore has size at least *opt* and is thus an upper bound to *opt*. The proof works analogously for G_{row}. □

See Figure 4.1 for an example consisting of a product graph complement and the column contracted graph.

Computing the contracted graph has a running time complexity identical to the previous approach, however, in this case the search for pairs of columns forming cliques is not restricted to columns with identical label. Consider Algorithm 1: We can modify it to construct a column contracted graph with n vertices. To build the contracted graph, we simply have to remove the label condition from Line 17 and replace Lines 19 - 21 with the creation of an edge between vertices i and j.

This method works regardless of the labels present, however, the computed upper bound may be larger than the original bound. Therefore, we propose using this method only in conjunction with some other method of arriving at an upper bound, such as Equation 4.1.

Even though computing a maximum independent set is NP-hard [GJ78, GJ90], the contracted graphs are likely to be easily solved. This is due to the fact, that the contracted graphs are likely to contain very few edges and a lot of isolated nodes. We expect it to be a rare occurrence to have a large number of fully connected columns, as the product graph complement would have to be extremely dense. Additionally, the contracted graphs are rather small in comparison to the product graph complement, as the number of vertices is equal to the number of vertices of the input graphs. We experiment on this method in Section 5.5.

Note, that both upper bounds can be modified to work in conjunction with the product graph as well: Instead of looking for fully connected columns we instead simply have to consider *independent* columns and then proceed as described.

4.2 Mapping Vertices Directly

If before the actual execution of an MCS algorithm we are able to directly map vertices of the two input graphs onto each other, we are thus able to decrease the problem size and hence simplify the problem. Here, we describe an approach which allows for such mappings for vertices with special properties.

Lemma 4.2.1. *Given the input graphs G_1 and G_2, if there are two vertices $u \in V(G_1)$ and $v \in V(G_2)$ with $\mu_{G_1}(u) = \mu_{G_2}(v)$ and $\deg(u) = \deg(v) = 0$, u can be mapped to v directly, and thus both vertices can be excluded when building the product graph (-complement).*

Proof. Mapping an isolated vertex $u \in V(G_1)$ to any vertex $v \in V(G_2)$ immediately excludes all neighbors $w \in N(v)$ from the solution, as no neighbor of v can be added to the solution without violating the maximum common subgraph conditions. Therefore, if both u and v are isolated, u can be mapped to v without loss of optimality. □

This direct mapping is valid for the maximum common *induced* subgraph problem. If we consider the maximum common *connected* subgraph problem, either u and v can be removed from the solution and excluded from the product graph (-complement) altogether, or - if the resulting maximum common subgraph would be empty - u and v are the only mapping in the solution.

Lemma 4.2.2. *Given a vertex $u \in V(G_1)$, such that the label $\mu_{G_1}(u)$ does not exist in G_2, u can be removed from G_1.*

Proof. Given that no vertex $v \in V(G_2)$ exists, such that $\mu_{G_1}(u) = \mu_{G_2}(v)$, u cannot be mapped onto any vertex of G_2. Therefore, u can be removed from G_1. □

We consider a vertex $u \in V(G_1)$ to be *effectively isolated*, if $\forall w \in N(u) : \mu_{G_1}(w) \notin \mu_{G_2}(V(G_2))$. In other words, if a vertex $u \in V(G_1)$ is only adjacent to vertices with labels that do not exist in G_2, u is effectively isolated, as all of its neighbors may be removed from G_1.

Based on Lemma 4.2.1, Lemma 4.2.2 and the definition of *effectively isolated*, we arrive at the following Corollary:

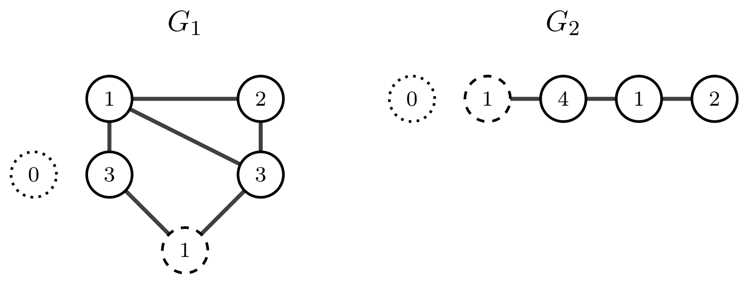


Figure 4.2: Example where one vertex-pair can be mapped directly due to being isolated (dotted vertices) and one vertex-pair due to being *effectively* isolated (dashed vertices). The vertices are labeled with $l \in \{0, 1, 2, 3, 4\}$.

Corollary 4.2.1. *Given the input graphs G_1 and G_2, if there are two vertices $u \in V(G_1)$ and $v \in V(G_2)$ with $\mu_{G_1}(u) = \mu_{G_2}(v)$ and both u and v are effectively isolated, u can be mapped to v directly.*

An example for both the mapping of isolated and effectively isolated vertices can be seen in Figure 4.2.
Let $n_{G_1} = |V(G_1)|$, $m_{G_1} = |E(G_1)|$ and analogously for G_2, finding all such isolated or effectively isolated vertices can be done in $\mathcal{O}(m_{G_1} + m_{G_2})$, assuming that it is known, which labels exist in both graphs. This is done by iterating over all outgoing edges of a vertex, to determine whether the label of a neighboring vertex exists in the other graph or not. Doing this for all vertices of both graphs requires iterating over the edges of both graphs at most twice. The iteration over the edges of a vertex can be stopped as soon as a vertex has an effective degree ≥ 1, as the vertex is therefore not effectively isolated.

4.3 Domination Reduction

As described in Section 3.3, KaMIS employs a set of reduction rules in order to reduce the graph. One such reduction rule is the *domination* rule, first described by Fomin et al. [FGK09]. A vertex u is said to dominate a vertex v, if $N[v] \subseteq N[u]$. In that case, u can be removed from the graph, as it can be replaced by v for any maximum independent set [FGK09, LSS+19].
If for two vertices $u, u' \in V(G_1)$ and for all $v \in V(G_2)$ uv dominates $u'v$ in the product graph complement, the domination reduction will remove all uv. Therefore, after applying all domination reductions, u is no longer represented in the product graph complement and can thus be removed from G_1 entirely.
If we are able to find such relationships based entirely on the input graphs alone, this may first of all help to reduce the upper bound and secondly, it may open up new possibilities for the direct mapping discussed in Section 4.2. However, we first need to determine how to detect such domination relations *before* constructing the product graph complement. We next describe rules which can detect *some* occurrences of such a domination relation.

Lemma 4.3.1. *Given two vertices $u, u' \in V(G_1)$ such that $\mu_{G_1}(u) = \mu_{G_1}(u')$, if u dominates u' and $\forall v \in V(G_2) : \mu_{G_2}(v) = \mu_{G_1}(u)$, and the labels of the neighbors of v are different to the labels of the neighbors of u, uv dominates $u'v$ in the product graph complement $\forall v \in V(G_2)$ and thus u can be removed from G_1.*

Proof. If for two vertices $u, u' \in V(G_1)$ u dominates u', we have $N[u] \supseteq N[u']$ and therefore, in ${G_1}^{\mathcal{C}}$ $N(u') \supseteq N(u)$. Given the condition that for every $v \in V(G_2)$ which has the same label as u and u', the labels of the neighbors of v have to be different to the labels of the neighbors of u, for any edge $\{u, w\}$ with $w \in N(u) \setminus N(v)$, no similar edge can exist in G_2, i.e. an edge where one endpoint has label $\mu_{G_1}(u)$ and the other endpoint has label $\mu_{G_1}(w)$. Therefore, computing ${G_1}^{\mathcal{C}} \times G_2$ cannot create edges for $u'v'$ with $v' \in V(G_2)$ which are at the same time not inserted for uv' as well. In other words, for edges $\{u', w\} \in {G_1}^{\mathcal{C}}$ such that $w \in N(u) \setminus N(v)$, the computation of ${G_1}^{\mathcal{C}} \times G_2$ cannot create any edges for the product graph complement Since $N[u] \supseteq N[u']$, computing $G_1 \times {G_2}^{\mathcal{C}}$ inserts for u at least the same edges as for u', and maybe some additional edges as well. Finally, adding the row and column cliques creates the same edges for both uv and $u'v$, $\forall v \in V(G_2)$. Putting everything together, given the conditions of Lemma 4.3.1: $\forall v \in V(G_2) : N[uv] \supseteq N[u'v]$, which means that all uv can be removed from $PG^{\mathcal{C}}$, and therefore u can be removed from G_1. □

Therefore, we can compute such cases by computing dominations in the input graphs and checking, if the condition is met. Computing which vertices v are dominated by a vertex u requires comparing the neighborhoods of u and v. To do this, we iterate over the neighbors v of u and check for each neighbor w of v, if w is a neighbor of u as well. This has a worst-case running time complexity of $\mathcal{O}(\deg(u)\deg(v))$, doing this for each vertex of a graph results in an overall running time complexity of $\mathcal{O}(nm)$. After determining the set of vertices dominated by $u \in V(G_1)$, we next need to compare the labels of the neighborhood of u with the labels of the neighborhood of all vertices $v \in V(G_2)$ with $\mu_{G_1}(u) = \mu_{G_2}(v)$. There can be at most $n_{G_2} = |V(G_2)|$ such vertices, for each vertex we compute the intersection of neighbor labels. To speed up this process, we construct a list of frequencies of the labels of the neighbors of a vertex in advance. Therefore, computing the intersection of the neighbor labels then simply consists of comparing these lists of label frequencies, which requires time linear in the number L of existing labels. Combining the computational effort for a single domination relationship results in a running time complexity of $\mathcal{O}(Ln_{G_2})$. Putting everything together, to determine which vertices of G_1 can be reduced we arrive at a complexity of $\mathcal{O}(n_{G_1}m_{G_1} + n_{G_1}^2 Ln_{G_2})$, as a vertex $u \in V(G_1)$ may dominate in the worst case n_{G_1} vertices.
Note, that this reduction can be applied to the input graphs before running any of the approaches discussed for solving for MCS.

4.4 Exploiting Automorphisms

As mentioned in Section 3.1.1, Koch [Koc01] notes that automorphisms frequently occur in product graphs. However, to the best of our knowledge, nowhere in the literature it has been attempted to exploit these automorphisms in order to reduce the search space. We thus in the following discuss how automorphisms may be exploited in the context of MCS based on *orbits* (cf. Section 3.2). We first discuss approaches to exploit automorphisms in McSplit(+RL), followed by the more general product graph and product graph complement. In the following we will use *orbit*(v) to denote the orbit in which a vertex v is located.

4.4.1 Automorphisms for McSplit and McSplit+RL

For McSplit and McSplit+RL we can exploit automorphisms for both input graphs, and thus potentially significantly decrease the search space. Unless otherwise specified, in the following we will only talk about McSplit, though everything applies to McSplit+RL as well.
A naïve method to exploit automorphisms is to simply compute the orbits of both input graphs once at the beginning. Next, at every branching step, we keep track of vertices of which orbits have already been explored at each branching state. We then simply skip over branches with vertices v, if for there is a any vertex v' which is located in the same orbit as v and at that branching state the branch of v' has been explored already.
Note, that this simple scheme does not guarantee correctness: Consider the graph depicted in Figure 4.3. We consider that graph to be one of the input graphs and we assume the other input graph to contain vertices with labels 0 and 1 as well. When for example vertex w is mapped, not

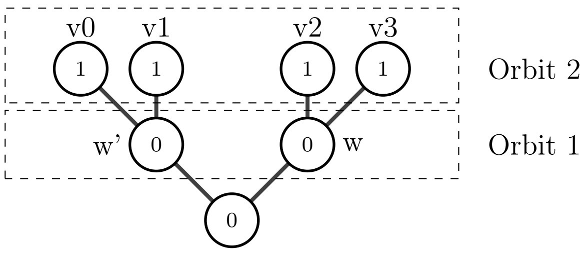


Figure 4.3: Example input graph used in illustrating possible problems with the exploitation of automorphisms.

all vertices of orbit 2 are still equivalent to another. Therefore, it is not correct to skip branching on vertex $v0$ simply because either vertex $v2$ or $v3$ where branched on already.

Recall, that McSplit changes the labels of vertices, when a vertex w is matched: vertices in the neighborhood of w have a 1 appended to their label, whereas vertices not in the neighborhood of w have a 0 appended to their label, and these updated labels partition the vertices into *domains*. We can alleviate the issue of incorrectness, by only skipping the branching for vertices which are in the same orbit *and* in the same *domain*. Therefore, in Figure 4.3, we would correctly differentiate between the vertices of orbit 2. With the new domains $v0$ and $v1$ would still be equivalent to another, and vertices $v2$ and $v3$ would still be equivalent to another. However, $v0$ and $v1$ would not be equivalent to $v2$ and $v3$ anymore. Note, that this approach only *alleviates* the aforementioned issue: Consider again the graph in Figure 4.3. If we copy and paste the graph, i.e. we have an input graph consisting of two identical connected components, we would have the vertices labeled 1 of the duplicate-component be equivalent to vertices $v0$ and $v1$, as they would still be in the same orbit and the same domain. However, these vertices are not equivalent to $v0$ and $v1$, as they have no matched vertex in their connected component. Therefore, this approach still does not guarantee correctness. We do, however, consider this approach in our experimental evaluation as a heuristic approach and compare its running time and result quality to other approaches.

To completely fix the aforementioned issues, we need to fully recompute the automorphisms at each branching state as to update the orbits to correctly reflect the new state of the input graphs. Note, that recomputing the automorphisms at every branch may be very costly w.r.t. running time. We therefore propose a heuristic algorithm for an update to the orbits using a modified Breadth-First-Search (BFS). Our update proceeds as follows: for each vertex we keep a list of distances to the current solution vertices. Whenever a new vertex-pair is added to the solution, we start a BFS from both vertices in their respective graphs. For any vertex u, that is encountered during that BFS, we add the distance of the solution vertex w to u to the distance list of u. Next, we hash that list of distances together with the original orbit number of u to arrive at a new orbit number. This algorithm is depicted in Algorithm 2.

Every time we go back up a level in the branching tree, we remove the last distance from the list of each vertex. For this algorithm we assume a collision-free hash function. Again consider the graph in Figure 4.3: With this algorithm, after mapping w, vertices $v0$ and $v1$ are still equivalent with a new orbit number and $v2$ and $v3$ are equivalent as well, again with a new orbit number. If we had again a duplicate of the graph as a second connected component, all other vertices with label 1 would remain unchanged in their original orbit.

Given the graph in Figure 4.4 we can clearly demonstrated the limitations of this approach: Consider vertices $v1$ and $v2$ are matched. The vertices of orbit 2 would be correctly separated due to the vertices residing in different domains. However, the vertices w and w' of orbit 1 would retain their equivalence with this update algorithm. Both vertices are non-adjacent to the vertices in the solution, thus their domains are equivalent, and the list of distances to the solution vertices is for both w and w' $\{2, 3\}$. Therefore, this heuristic would still treat w and w' as equivalent,

Algorithm 2: Pseudo-Code depicting the update of orbit association using a modified BFS

```
Data: $G, D, w$
Result: updated orbits for all vertices
1  $S \leftarrow \{w\}$ // Keeping track of which vertices we already visited
2  $Q \leftarrow \{w\}$ // Queue for BFS
3  // To compute the distance of a vertex $u$ from $w$
4  $\forall u \in V(G) : d_u \leftarrow 0$
5  while $v \in Q$ do
6      $Q \leftarrow Q \setminus \{v\}$
7      forall $u \in N(v) : u \notin S$ do
8          $d_u \leftarrow d_v + 1$
9          $Q \leftarrow Q \cup \{u\}$
10         $S \leftarrow S \cup \{u\}$
11     end
12     $D_v \leftarrow D_v \cup \{d_v\}$
13     $orbit'(v) \leftarrow$ hash($D_v$, orbit($v$))
14 end
15 return $orbit'$
```

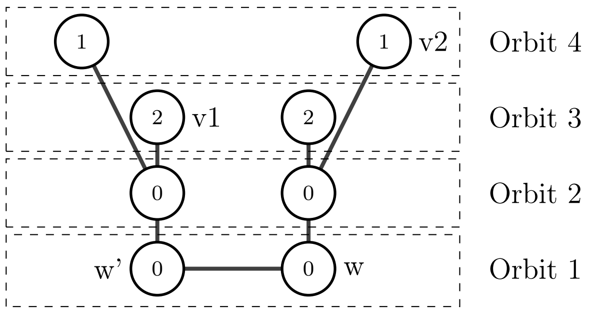


Figure 4.4: Example input graph illustrating a possible problem with the BFS orbit update

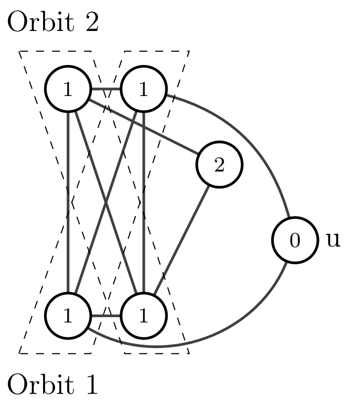


Figure 4.5: Example product graph complement with non-trivial orbits drawn in.

whereas the vertices are clearly not equivalent anymore.

For McSplit we introduced three different variants of exploiting automorphisms. First, the simple heuristic exploitation skips branching on a vertex v, if there is another vertex v', such that v and v' are in the same orbit and in the same domain and branching occurred on v' already. This approach requires overhead in time linear in the number of vertices of the input graphs at each branching step, however, out of our three approaches, this approach may lead to the most inexact automorphism exploitations. In the experimental evaluation we refer to this approach as *autom. heuristic*. Next, the approach utilizing a modified BFS to update orbit associations requires the computation of two BFS at each branching step and thus in the worst-case takes $\mathcal{O}(n+m)$ time for each input graph. However, this approach is a better approximation than the simple heuristic discussed before. We refer to this approach as *autom. bfs update* in our experimental evaluation. Finally, to obtain an exact exploitation of automorphisms, we suggest to recompute the orbits at each branching step at the expense of additional running time. In the experimental evaluation we refer to this approach as *autom. nauty recomp.* In Section 5.7 we compare these three variants with regards to running time as well as result quality.

4.4.2 Automorphisms of the Product Graph Complement

As demonstrated in the previous section, to exploit automorphisms correctly, we either need to fully recompute the automorphisms at each branching state or perform an update operation. Both methods can be applied to the product graph complement as well, in order to ensure optimality. However, we again discuss a simple heuristic approach, which we then experimentally evaluate in Section 5.7. We first describe some properties of orbits. Given two orbits i and j, there may be some edges between the vertices of the orbits or the orbits may be completely independent. Due to the orbits representing equivalence classes, it suffices to pick a single representative vertex u of orbit i and investigate if and how many edges u has to the vertices of orbit j. If that vertex u has no edges to the vertices of orbit j, no other vertex of orbit i can have edges to vertices of orbit j either. We can therefore efficiently determine the relationship between any two orbits.

Regarding our heuristic exploitation of automorphisms, we now only allow a branch to be skipped, if the orbit of the branching vertex is completely independent from the orbits of vertices currently in solution. Similar to the automorphism exploitation in McSplit(+RL), this constraint poses an improvement over the straight-forward skipping of branches such that fewer inexact exploitations occur. As an example consider the product graph complement in Figure 4.5: There are two non-trivial orbits as well as two trivial orbits. Assume that vertex u is put into the solution. As orbit 2 is independent of the trivial orbit of vertex u, we only branch on one vertex of orbit 2 at that branching state.

To use this scheme, we first need to determine the relationships between orbits, which in the

worst-case requires iterating over all edges of the product graph complement, therefore has worst-case complexity of $\mathcal{O}(m)$. During the branching, we need to look up the relationship between the orbit of a branching vertex u and the orbits of vertices currently in solution. This can be done in time linear in the size of the solution.
Note, that a similar approach can be applied to the product graph: Instead of requiring orbits to be independent, we require orbits to be fully connected. This condition can again be determined in worst-case time $\mathcal{O}(m)$, as it again suffices to pick a representative of an orbit and check its edges. In Section 5.7 we evaluate the running time and result quality of this heuristic approach.

4.5 Initial Scores for Reinforcement Learning

With every instantiation of McSplit+RL on a pair of input graphs, the algorithm starts off with an initial score of 0 for each vertex. In this section we describe two different approaches which may be used to assign an initial score to vertices, based on information about their neighborhood and the likeness of vertices to the vertices of the other input graph.
We first discuss an approach utilizing the Weisfeiler-Leman algorithm. In our approach, we modify the Weisfeiler-Leman algorithm to run for up to k iterations or until a stable labeling is achieved. This yields a set of up to $k+1$ labels for all vertices of both graphs, up to k labels from Weisfeiler-Leman plus one initial label. Let L_j denote the number of vertices of G_2 with label j and let $\lambda(u)_i$ denote the i-th label of a vertex $u \in V(G_1)$ w.r.t. to the Weisfeiler-Leman algorithm. Then the *score* s_u of a vertex u is computed as follows:

$$s_u = \sum_i L_{\lambda(u)_i} w_i \tag{4.2}$$

where w_i denotes a weight parameter, by which the score of an iteration is weighted. The score of a vertex u is calculated by a weighted sum of the number of vertices with matching label in the other input graph. As matching labels in the i-th iteration indicate identical i-hop neighborhood, it seems only natural to increase the weight parameter with the iteration, we suggest setting $w_i = 2^i$. Note, that Weisfeiler-Leman requires identical neighborhoods in order to result in the same label. However, in reality it is more likely, that the neighborhoods are similar, but not identical. Therefore, we suggest an additional approach based on the *Jaccard-Index*, which computes a score for the similarity of sets. The Jaccard-Index is defined on two sets A and B as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.3}$$

i.e. the size of the intersection of the two sets normalized by the size of their union [Jac12]. Thus the Jaccard-Index is a real number between 0 and 1.
For our application, we can choose the sets A and B to be multi-sets containing the labels of the closed neighborhood of two vertices and thus compute a similarity measure for two vertices. However, we can take it a step further and allow this similarity computation to take place over a number of iterations and by that consider not only the immediate neighborhood, but the neighborhood of the neighbors as well. To accomplish this, we start off by encoding the label of each vertex u into a vector $x_u^{(0)}$ of size L, where L denotes the number of existing labels in both input graphs. We only set the vector entry corresponding to the label of a vertex to 1, all other entries are set to 0 (this is called a *one-hot encoding*). Next, for every iteration $i \in [1..j]$, we compute a new vector $x_u^{(i)}$ for each vertex u, which is computed as $\sum_{\forall v \in N(u)} x_v^{(i-1)}$, i.e. the summation of the vectors of the previous iteration of all of u's neighbors. After each iteration we update the score s_u of u with these vectors using a slightly modified definition of the Jaccard-Index, as we are working on vectors here and not on sets:

$$J_W(x_u, x_v) = \frac{\sum_l \min(x_u[l], x_v[l])}{\sum_l \max(x_u[l], x_v[l])} \tag{4.4}$$

where $x_u[l]$ represents the l-th element of the vector x_u. This modified Jaccard-Index is known as the *Ruzicka similarity* [War16]. See Algorithm 3 for a pseudo-code representation of the algorithm.

Algorithm 3: Pseudo-Code depicting the computation of initial scores for McSplit+RL using the Jaccard-Index

```
Data: $G_1, G_2, k$
Result: initial scores $s$ for all vertices
// vector of size $L$, where $L$ denotes the number of labels in the input graphs
$\forall u \in V(G_1) : x_u^{(0)} \leftarrow \langle 0 \cdots 0 \rangle$
$\forall v \in V(G_2) : x_v^{(0)} \leftarrow \langle 0 \cdots 0 \rangle$
forall $u \in V(G_1)$ do
    $s_u \leftarrow 0$
    $x_u^{(0)}[\mu_{G_1}(u)] \leftarrow 1$
end
forall $v \in V(G_2)$ do
    $s_v \leftarrow 0$
    $x_v^{(0)}[\mu_{G_2}(v)] \leftarrow 1$
end
for $i = 1$ to $k$ do
    forall $u \in V(G_1)$ do
        $x_u^{(i)} \leftarrow \sum_{\forall u' \in N(u)} x_{u'}^{(i-1)}$
    end
    forall $v \in V(G_2)$ do
        $x_v^{(i)} \leftarrow \sum_{\forall v' \in N(v)} x_{v'}^{(i-1)}$
    end
    forall $u \in V(G_1)$ do
        forall $v \in V(G_2) : \mu_{G_1}(u) = \mu_{G_2}(v)$ do
            $s_u \leftarrow s_u + J_W(x_u^{(i)}, x_v^{(i)})$
            $s_v \leftarrow s_v + J_W(x_u^{(i)}, x_v^{(i)})$
        end
    end
end
return $s$
```

Using either the Weisfeiler-Leman or our Jaccard method we arrive at initial scores for each vertex, which are dependent on the similarity of a vertex to vertices from the other input graph.

4.6 Speeding up KaMIS for MCS

In this section we discuss some modifications we propose for KaMIS [LSS+19], as to optimize for the MCS problem setting. We first discuss a modification of the Local Search heuristic solver of KaMIS. As discussed in Section 3.3, the Local Search performs a fixed number of iterations, where the algorithm tries to improve the current solution. By default, KaMIS performs 1 million such iterations, which can be very time consuming. We therefore propose a modification, which limits the number of iterations based on when the solution was last increased.

Our first approach is a simple *improvement limit* T, which dictates the maximum number of iterations taken after the solution has been updated to a larger solution. If within these T iterations a larger solution is found, the algorithm again has a budget of T iterations to further increase the solution. Once the T iterations have been exhausted, the algorithm is forced to terminate. This fixed size limit may still cause a large number of iterations performed on simpler instances, without any improvement. We therefore additionally introduce an *adaptive limit* A, with $A \lll T$. As with T, the algorithm is limited to A iterations after finding a larger solution, however, with the adaptive limit we double A every time a larger solution is found, as the algorithm is more likely to require more steps to find even larger solutions. To limit the size of A, A can be increased to at most T. This approach has the benefit of keeping the number of iterations taken smaller for simpler instances, where only few improvement steps are needed. For an experimental evaluation of these limits see Section 5.4.1.

Next, we propose a change to the reductions employed by the Branch-And-Reduce solver throughout execution as well as by the Local Search at the beginning. In its original form, KaMIS offers two different *reduction styles*: One style for dense graphs and one style for sparse graphs. These two styles differ in which reductions are applied to the graph, with the sparse style being a super-set of the dense style. However, some reductions may be cost prohibitive w.r.t. running time in the context of MCS. Additionally, some of the reductions used in KaMIS rely on the weight of a vertex being larger than the weight of its neighborhood (*neighborhood* reduction), which can never apply in our unweighted product graph complement and therefore we can safely disable such reductions. Next, we note that the order in which the reductions are applied matters: KaMIS keeps a list of the reductions, attempting to apply them one after the other. As soon as some reduction was applied successfully, KaMIS restarts at the beginning of the list. Thus reductions at the front of the list will be tried more often. In preliminary experiments we notice that the *domination* reduction applies successfully most often, therefore we propose to put that reduction to the front of the list. Next in the list, we put the *clique* reduction, followed by the *fold* and *twin* reductions. See Section 5.4.2 for an experimental evaluation of our reduction style versus the original two styles.

4.7 Summary

Our contributions presented in this chapter are manifold, though some of the techniques may only be applied to specific solvers. Table 4.1 gives a brief summary of our contributions, showing for which kind of algorithms our techniques may apply. We presented two techniques, which are algorithm-agnostic and as such may be applied to any approach, namely the mapping of vertices and the domination reduction. Both techniques can simply be applied to the input graphs before the actual MCS algorithm is run. Other techniques are tailored to given solvers, such as the initial scores for McSplit+RL, the reduction style for KaMIS solvers or the iteration limit for the Local Search solver of KaMIS. Finally, the exploitation of automorphisms may be applied to many solvers, in this work we presented means of exploiting automorphisms for solvers operating on the product graph (-complement) as well as for McSplit(+RL).

Suggested Technique	Applicable Solvers	Reference
Improved upper bounds	any solvers operating on the product graph (-complement)	Section 4.1
Mapping of vertices	any solver	Section 4.2
Domination reduction	any solver	Section 4.3
Exploitation of automorphisms	McSplit(+RL) as well as solvers operating on the product graph (-complement)	Section 4.4
Initial scores	McSplit+RL	Section 4.5
KaMIS reduction style	Branch-And-Reduce and Local Search solvers of KaMIS	Section 4.6
Iteration Limit	Local Search solver of KaMIS	Section 4.6

Table 4.1: An overview of the techniques suggested in this chapter and to which existing algorithm each technique may be applied.

5 Experimental Evaluation

In this chapter we evaluate our approaches experimentally and compare the different approaches. The remainder of this chapter is structured as follows. First, we describe our methodology and experimental environment. Next, we discuss the instances, on which we run our experiments, followed by an analysis of the product graph complement densities for the instances used. We follow by evaluating our changes made to the KaMIS solvers specifically for the MCS problem, as well as an evaluation of the tight bound and domination reduction. Next, we evaluate the initial scoring methods for McSplit+RL and evaluate the performance against the baseline algorithm. Following that, we discuss the exploitation of automorphisms, both for McSplit (McSplit+RL) as well as for the Branch-And-Reduce solver of KaMIS. Finally, we perform an overall comparison of the best competitors found in the aforementioned evaluations and evaluate their running time and result quality performance against a state-of-the-art clique solver.

5.1 Environment and Methodology

For our experiments, we utilize an Ubuntu `18.04.5` machine running kernel version `4.15.0-106-generic`. The machine consists of eight `AMD Opteron™ 6174` processors, with six cores each. Each processor has 512 kB and 5118 kB of L2 and L3 cache, respectively. The machine has a NUMA architecture, meaning that out of the 256 GB of RAM available in total, each processor only has 32 GB of RAM available locally. To prevent the added cost of accessing non-local memory, all experiments are pinned to a single processor and its locally available memory.

In our experiments, we use the Local Search and Branch-And-Reduce solvers from KaMIS [LSS+19] [1] in version `2.0`, McSplit [MPT17] [2], McSplit+RL [LLJH20] [3] as well as the clique solver MoMC [LJM17] [4]. For computing the automorphisms, we utilize `nauty` [MP14] [5] in version `2.7r1`. All code is written in C++11 and compiled with gcc 7.5.0 with full optimization enabled (O3). The only exception is `nauty`, which is written in C.

We run each experiment once, as the differences in running time would not be sufficiently large as to influence the overall pictures. We only report the running times of the algorithms, IO times are thus not included in the running time. All experiments are given a maximum of 1000 seconds time, after which the best result found so far is reported. To compare the various approaches, we compare the cumulative number of instances solved within 1000 seconds as well as the result quality obtained.

5.2 Instances

Following the methodology of McCreesh et al. [MPT17], we experiment on sets of randomly generated subgraph isomorphism graphs due to Santo et al. [SFSV03] and Conte et al. [CFV07]. These instances are grouped into two categories: Unlabeled and undirected instances (`mcsplain`), as well as directed, edge- and vertex labeled instances (`mcsved`). All of these instances can be

[1] `https://github.com/KarlsruheMIS/KaMIS`
[2] `https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph`
[3] `https://github.com/JHL-HUST/McSplit-RL`
[4] `https://home.mis.u-picardie.fr/~cli/EnglishPage.html`
[5] `https://pallini.di.uniroma1.it/`

Instance Group	min n	max n	avg. n	min m	max m	avg. m	avg. density
AIDS	2	95	15.92	2	206	32.90	2.02
COX2	32	56	41.24	68	118	86.91	2.11
proteins	4	620	38.57	10	2098	143.56	3.73
mcsplain	15	50	35.00	38	898	181.12	5.01
mcsved	40	100	75.40	59	450	167.76	2.16

Table 5.1: Properties of the instances used in the experimental evaluation.

found in the McSplit code repository [6], along with predefined pairings of input graphs to use as the inputs to experiments. For our experiments, we run the first 500 of these instances of both categories. We additionally include real-world instances sourced from *TUDatasets* [MKB$^+$20], which are undirected and vertex labeled. These instances represent `proteins` and small molecules, where `AIDS` represents molecules from research for fighting against the disease and `COX2` represent molecules, which bind to *cyclooxygenase* enzymes. As with the McSplit instances, we run experiments on the first 500 of these pairs. An exception are the `COX2` instances, where we only have a list of 233 instance pairs, due to the smaller number of graphs.
See Table 5.1 for an overview of the instances attributes. We can clearly see that `proteins` has the largest overall instance by far in terms of number of vertices and number of edges, whereas on average the largest instances are in the `mcsved` set with regards to number of vertices. `AIDS` has some very small instances, consisting of only 2 vertices and on average the instances are the smallest as well. The `mcsplain` instances have the highest density. All `COX2` are fairly similar in size, with the number of vertices only ranging from 32 to 56.

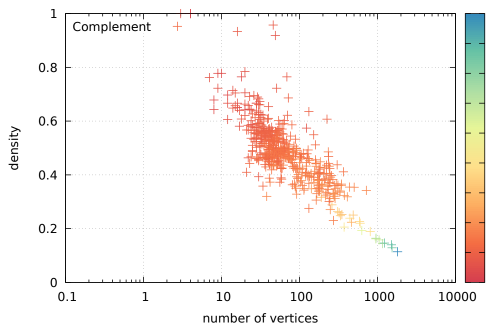
5.3 Product Graph Complement Density

We first investigate the densities of the product graph complement, as the density heavily influences the Branch-And-Reduce and Local Search solvers. Figure 5.1 shows the density for all five instance sets, along with a color-encoding of the overall maximum result found for these instances. The density is computed as the number of existing edges divided by the maximum number of possible edges. We first of all observe a negative correlation between the density and the number of vertices: For all instance sets there is a downward trend of the density with increasing number of vertices. Next, we observe that the two instance sets `AIDS` and `proteins` have product graph complement graphs which widely vary in density. For both sets, some instances lead to almost complete graphs, whereas some instances have a density of 0.1 or less. The densities of `COX2` and `mcsved` instances vary fairly little, most graphs have a density between 0.05 and 0.3. And finally, the `mcsplain` instance set is somewhere in between, the density ranges from about 0.15 to 0.55. With the color encoding of the maximum solution size found overall, we additionally observe a negative correlation between solution size and number of vertices: With larger number of vertices, the solution size increases as well. Additionally, for `mcsplain` we can observe that the density negatively corresponds to the solution size, with sparser instances reaching larger solutions.

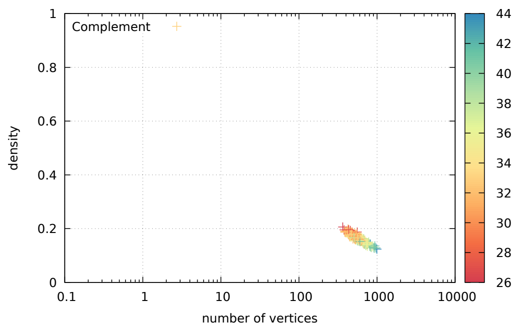
5.4 KaMIS Improvements

In this section we experimentally evaluate the improvements made specifically to KaMIS, as discussed in Section 4.6. We start by comparing various iteration limits with the default limit of

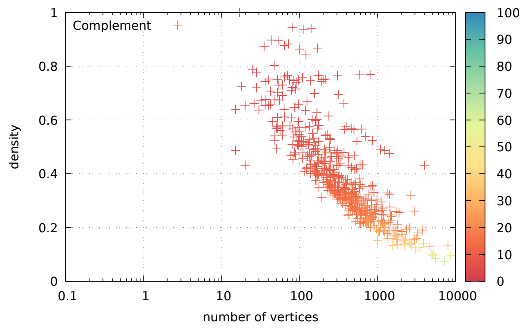
[6] `https://github.com/jamestrimble/ijcai2017-partitioning-common-subgraph`



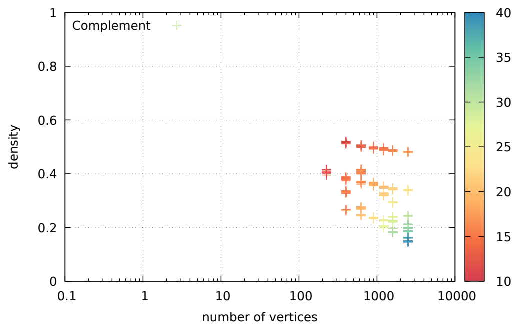
(a) AIDS



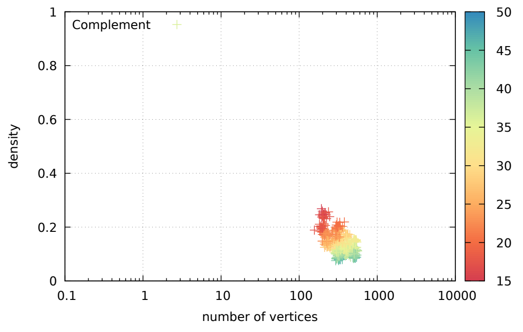
(b) COX2



(c) proteins



(d) mcsplain



(e) mcsved

Figure 5.1: Density of the product graph complement graphs for all instance sets.

one million iterations for two data sets for the Local Search solver. We follow by comparing our new reduction style with the default styles for the Branch-And-Reduce solver.

5.4.1 Local Search Iteration Limit

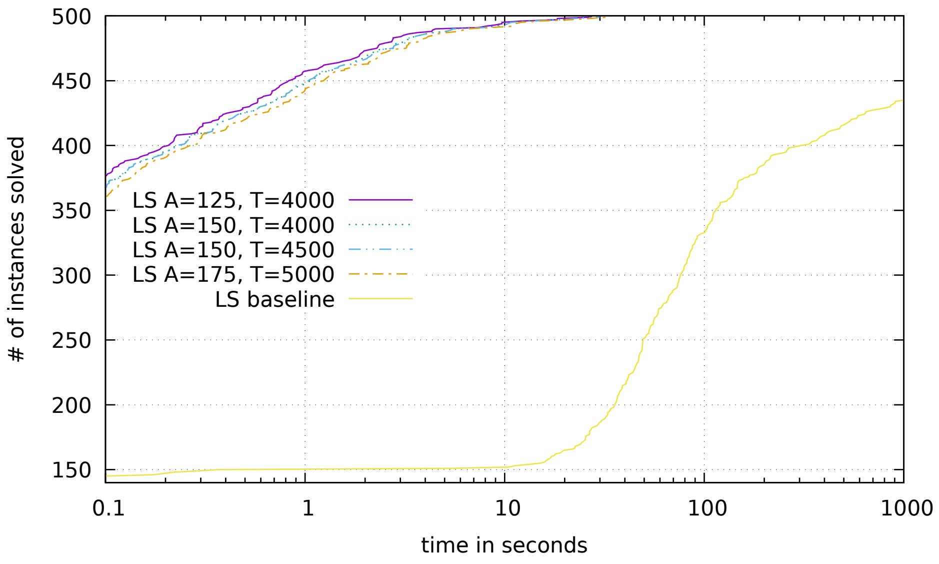
We first compare the iteration limits. For this, we experiment with various configurations for all instance sets, with the aim to find a good configuration, which provides a suitable trade-off between speedup and result quality, when compared to the baseline Local Search algorithm.

For `AIDS` instances, we experimented on various configurations based on some preliminary experiments, starting with $A = 125$ and $T = 4\,000$ up to $A = 175$ and $T = 5\,000$. Figure 5.2a gives a comparison of the running time of the various configurations and the baseline algorithm. On the y-axis we have the cumulative number of instances solved and on the logarithmic x-axis we see the running time in seconds. As we can see in Figure 5.2a, all of these configurations with iteration limits significantly outperform the default Local Search behavior in terms of instances solved over time, though the difference among the four configurations is comparatively small. However, when we look at Figure 5.2b, we can observe a difference in the result quality. This figure shows how well a given approach has solved the instances, relative to the maximum solution achieved by any approach we investigate. On the x-axis we have a relative distance t to the maximum result MAX and on the y-axis we see the relative number of instances that were solved to at least $t \times MAX$. The further to the top left any approach is, the better.

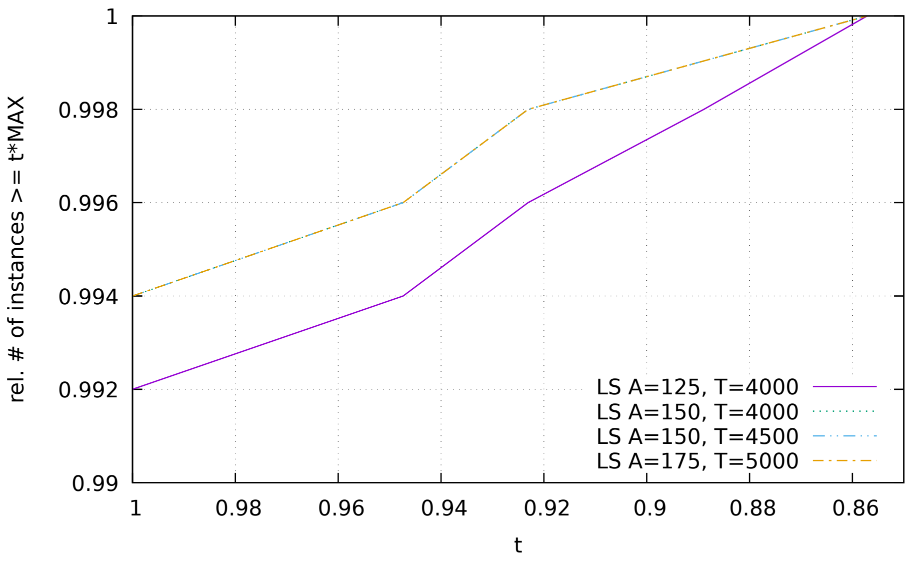
With respect to result quality (Figure 5.2b) the baseline version of the Local Search achieves the overall maximum result on all instances, therefore it would only be represented as a dot in the top left corner. Comparing the various configurations with iteration limits, we see that the configuration with $A = 125$ and $T = 4\,000$ performs slightly worse, whereas the other three configurations yield the exact same results. However, considering we overall ran on 500 instances, achieving the maximum on 99.2% of these instances means that for only four instances this configuration yielded a smaller result, whereas for 99.4% we find the maximum solution for 497 instances and only three instances are solved worse. In the worst-case, the result quality drops down to roughly 86% of the maximum solution found. Given how close all configurations are in terms of running time, we therefore conclude that the configuration with $A = 150$ and $T = 4\,000$ yields the best trade-off in terms of speed and result quality.

Next, we look at `COX2` instances, where finding good configurations has proven to be challenging. Remember, that the default behavior of the Local Search algorithm is to perform one million iterations. As we can see in Figure 5.3b, an increase of T to $500\,000$ and A to $15\,000$ is necessary in order to get reasonably close to the result quality of the baseline algorithm. Even then, there are overall still four instances, where the baseline algorithm finds larger results. For $T = 200\,000$ and $A = 6\,000$, we observe overall eight instances, where the result quality is smaller than that found by the baseline algorithm. Further down, with $T = 100\,000$ and $A = 3\,000$ there are 14 such instances. Regarding the running time, we can see in Figure 5.3a that the baseline algorithm is very slow, taking over 100 seconds to solve most instances, whereas the variants with iteration limits perform better, though not as significantly as they did for `AIDS` instances. For our final evaluation we could use the configuration with $500\,000$ and A to $15\,000$, as the quality is very close to the baseline. However, the running time performance is still very poor. We thus propose the configuration with $T = 200\,000$ and $A = 6\,000$, as the running time is significantly better and the result quality only drops for further four instances compared to the other configuration.

For `proteins` instances, we experimented on configurations with T between $10\,000$ and $30\,000$ iterations. Similarly to `AIDS` instances, here we again see a significant improvement in running time (Figure 5.4a), whilst achieving similar or only slightly worse result quality (Figure 5.4b). More specifically, the baseline algorithm takes 100 seconds or more for most instances and the algorithm times out for more than half the instances. Whereas with the iteration limits, the algorithm only times out on a few instances and the overall running time performance is faster. Regarding the result quality, the configuration with $T = 30\,000$ and $A = 100$ leads to a smaller result on seven instances, when compared to the baseline. The next smallest configuration with

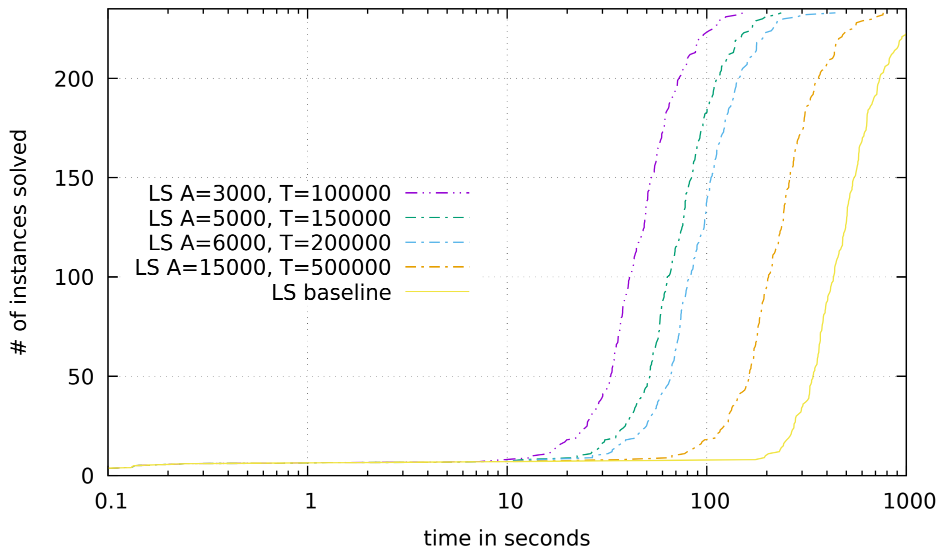


(a) Instances solved over time

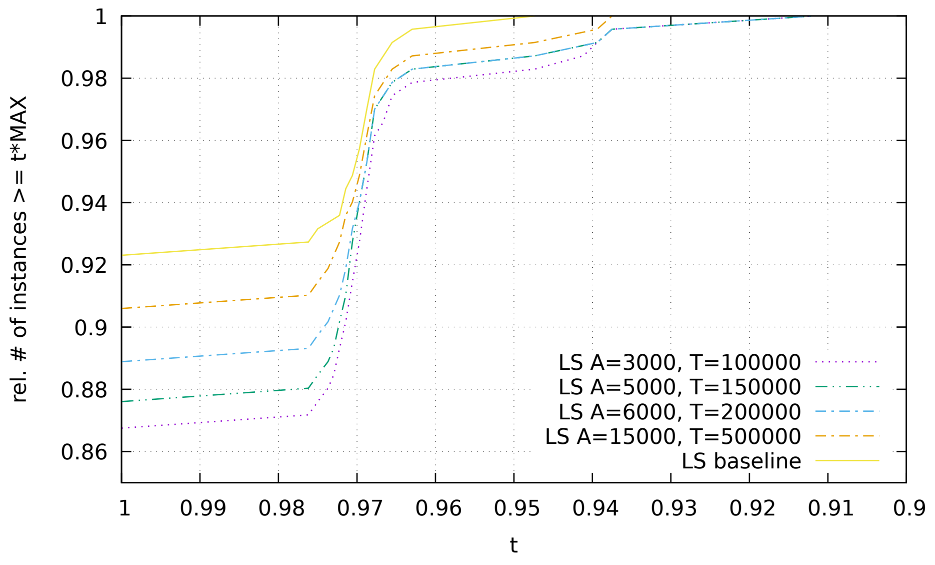


(b) Result quality

Figure 5.2: Comparison of running time and solution quality on `AIDS` instances for various different iteration limits for the Local Search solver.

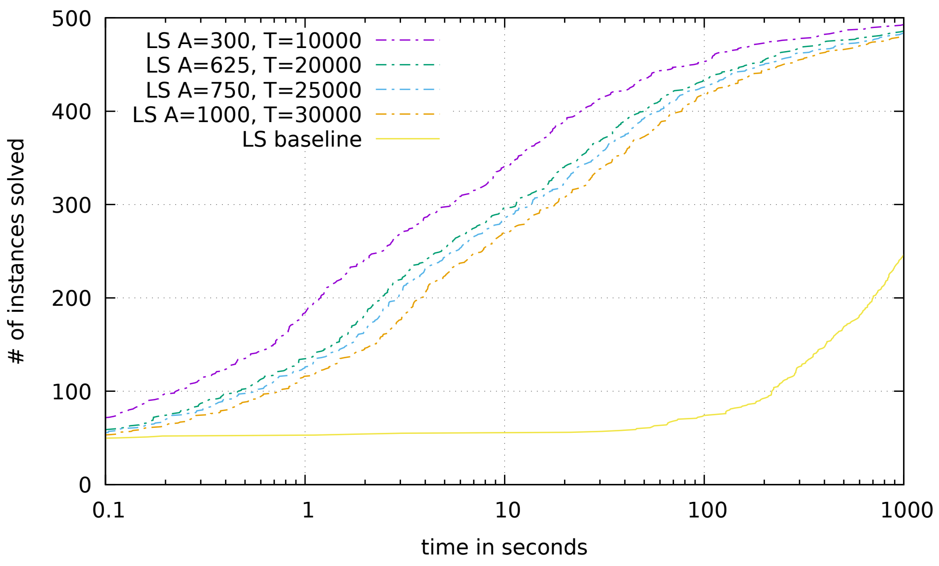


(a) Instances solved over time

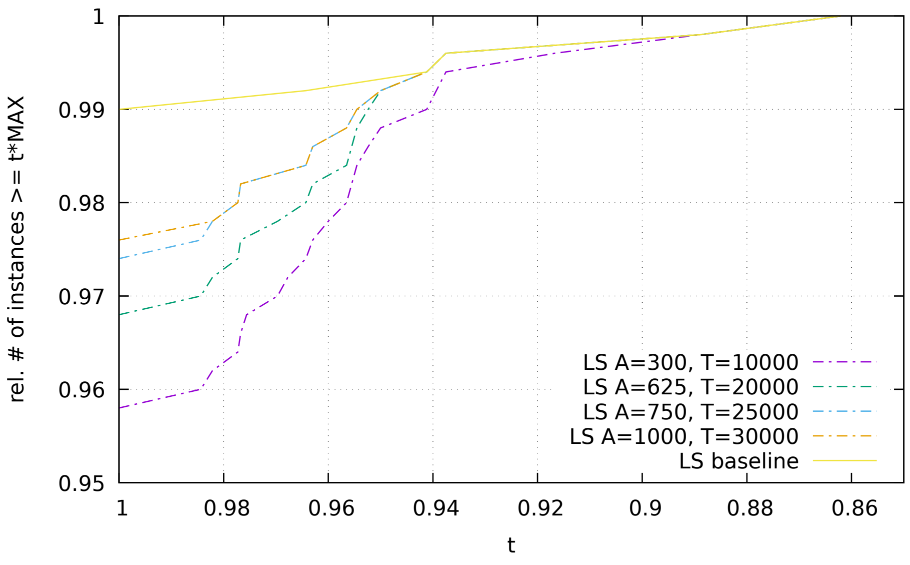


(b) Result quality

Figure 5.3: Comparison of running time and solution quality on `COX2` instances for various different iteration limits for the Local Search solver.



(a) Instances solved over time



(b) Result quality

Figure 5.4: Comparison of running time and solution quality on `proteins` instances for various different iteration limits for the Local Search solver.

Instance Group	A	T
AIDS	150	4 000
COX2	6 000	200 000
proteins	1 000	30 000
mcsplain	10 000	300 000
mcsved	1 250	40 000

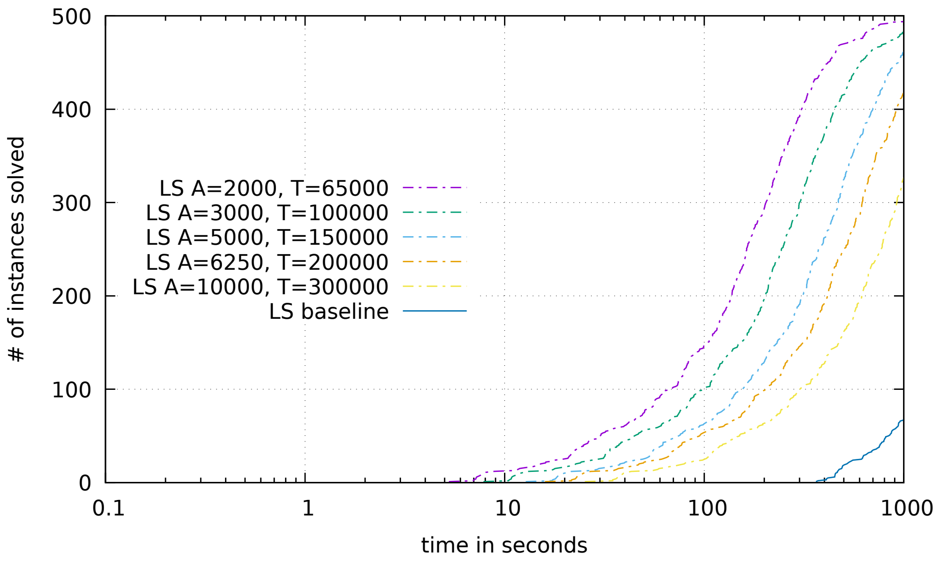
Table 5.2: Configurations used for Local Search throughout the remainder of the experimental evaluation.

$T = 25\,000$ and $A = 750$ there are eight instances, which result in a smaller solution size, for $T = 20\,000$ and $A = 625$ as well as $T = 10\,000$ and $A = 300$ it is 11 and 16 instances, respectively. For the overall comparison we thus chose the configuration $T = 30\,000$ and $A = 1000$, as it provides a good trade-off between speed and quality.

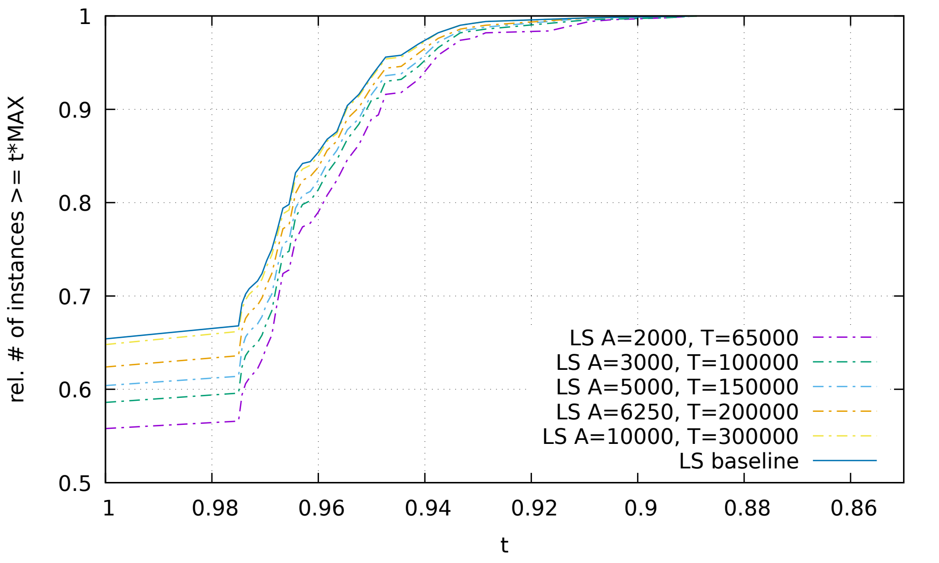
Next, we look at different configurations for the `mcsplain` instances. Here, preliminary experiments showed, that the number of iterations again has to be much higher than for `AIDS` instances. We therefore compare various configurations ranging from $T = 65\,000$ to $T = 300\,000$. Looking at Figure 5.5a, we immediately see, that the `mcsplain` instances pose a challenge to the Local Search, no instance is solved within less than five seconds, even for the smallest configuration. Additionally, the baseline algorithm requires at least 300 seconds per instance and only manages to solve less than 100 instances within the given timeout. If we now look at the result quality in Figure 5.5b, we see that the configuration with $T = 300\,000$ and $A = 10\,000$ performs very similarly to the baseline algorithm, in fact there are only three instances, for which this configuration achieves smaller results than the baseline algorithm. On the other end of the scale we have the configuration with $T = 65\,000$ and $A = 2\,000$, which performs significantly worse than the baseline algorithm, achieving smaller results for a total of 65 instances. For the other three configurations - with increasing T - the number of instances, for which that configuration achieves smaller results than the baseline, are 42, 31 and 18. This again indicates the difficulty of the `mcsplain` instances. Given these results, we opt to use the configuration, which provides the best result quality, which is the configuration with $T = 300\,000$ and $A = 10\,000$.

Finally, we consider various configurations for `mcsved` instances. Here, we experiment on configurations with T ranging from 15 000 to 50 000. If we first look the running time performance in Figure 5.6a, we again observe a significant boost in running time compared to the baseline algorithm. The various configurations are able to solve all instances within 20 seconds or less, whereas the baseline algorithm takes up to 750 seconds. Regarding the result quality, we see in Figure 5.6b, that the configuration with $T = 15\,000$ and $A = 500$ performs rather poorly in comparison, achieving a smaller result than the baseline on 18 instances. On the upper end, the configurations with $T = 50\,000$ and $A = 1\,500$ as well as $T = 40\,000$ and $A = 1\,250$ find smaller results on seven instances compared to the baseline. We therefore know, that increasing the limit T from $40,000$ to $50,000$ and A from 1 250 to 1 500 is not enough to increase the result quality. Therefore, we additionally tried a configuration of $T = 60\,000$ and $A = 2\,000$, however, the quality still remained unchanged. Given the significant boost in running time and the overall good result quality of the configuration $T = 40\,000$ and $A = 1\,250$, we choose this configuration for the overall evaluation.

Table 5.2 gives an overview of the configurations we decided on for the different instance groups. To confirm that our limits are not over fitted to the instances at hand, we performed additional experiments on previously unseen `AIDS` and `mcsplain` instances, where we compare the result quality and running time of the baseline algorithm with the chosen configuration for the instance set. Note, that we only run the Local Search solver on these unseen instances, therefore the overall maximum found solution is equal to the solution found by the baseline Local Search algorithm. First, we look at the comparison for `AIDS` instances between the chosen configuration of $T = 4\,000$

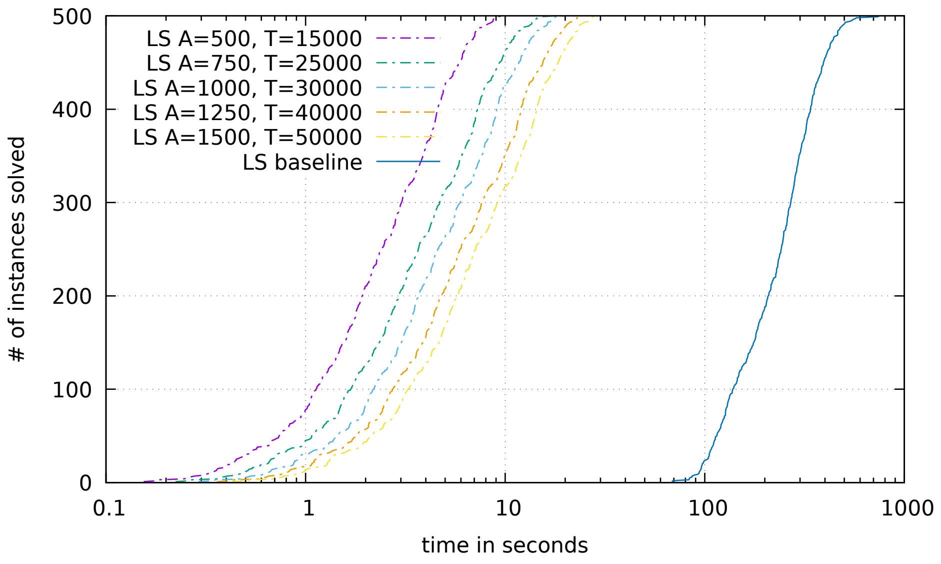


(a) Instances solved over time

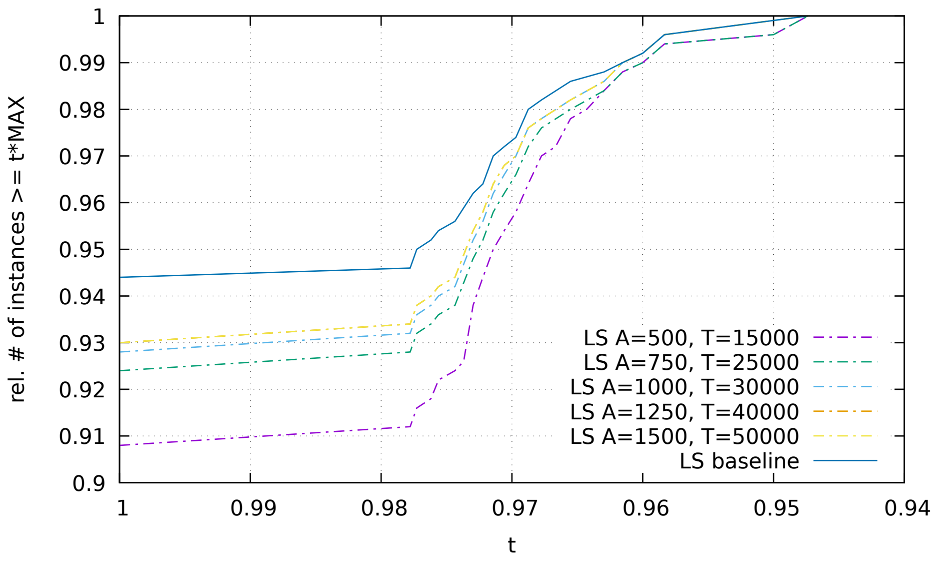


(b) Result quality

Figure 5.5: Comparison of running time and solution quality on `mcsplain` instances for various different iteration limits for the Local Search solver.

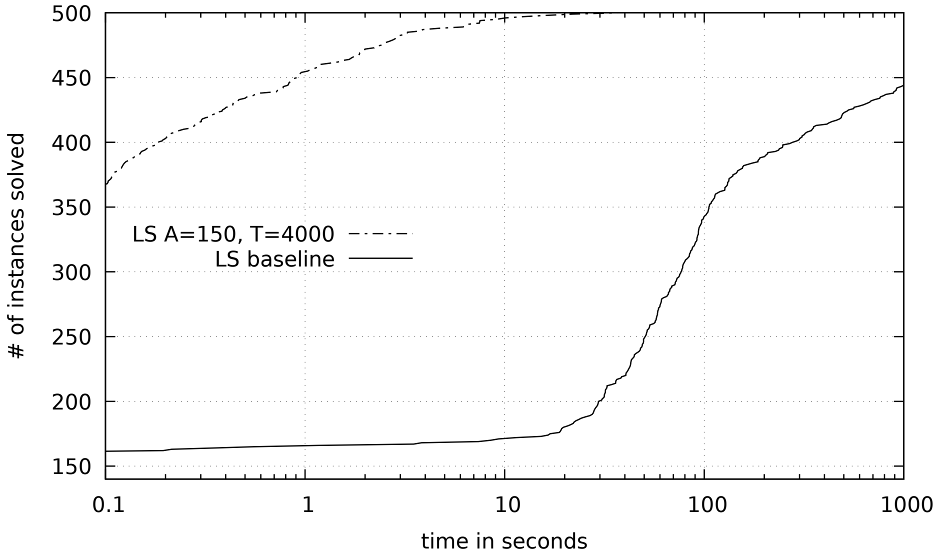


(a) Instances solved over time

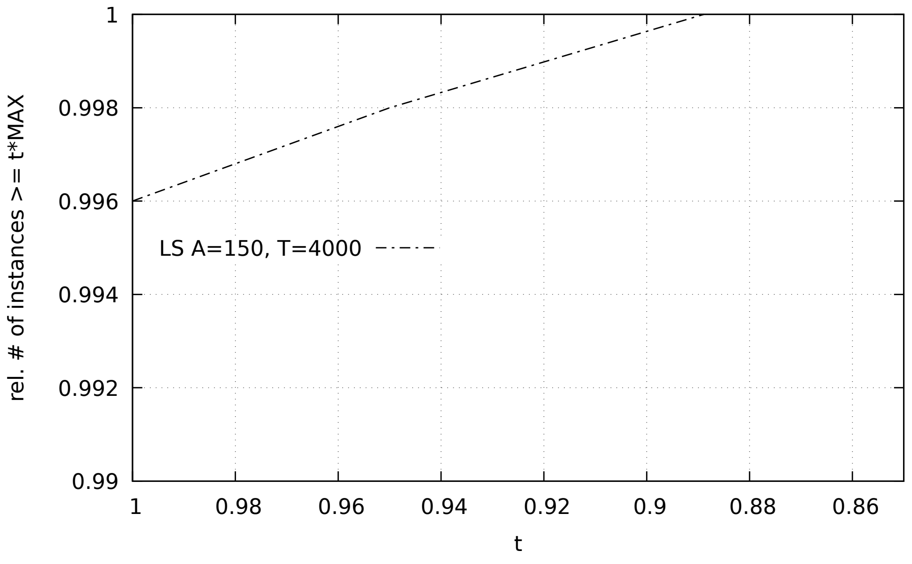


(b) Result quality

Figure 5.6: Comparison of running time and solution quality on `mcsved` instances for various different iteration limits for the Local Search solver.



(a) Instances solved over time



(b) Result quality

Figure 5.7: Comparison of running time and solution quality on previously unseen `AIDS` instances for the chosen iteration limit configuration versus the baseline algorithm of Local Search.

and $A = 150$ versus the baseline algorithm. We can see in Figure 5.7a, that the running time comparison looks almost identically to that of Figure 5.2a, with the variant with iteration limit significantly outperforming the baseline. Similarly, the picture painted by Figure 5.7b is very similar to that of Figure 5.2b, with the baseline algorithm achieving a larger result on only two instances. We have thus confirmed, that our chosen configuration for `AIDS` instances is not over fitted to the given data, but rather seems generally applicable to all instances from that set.

Next, we perform a similar comparison for `mcsplain` instances, where we again ran both the baseline and the configuration of $T = 300\,000$ and $A = 10\,000$ and compare their results. The running times of both variants are shown in Figure 5.8a, we again observe a drastic speed-up over the baseline algorithm. More importantly, though, is the result quality of the variant with iteration limits compared to the baseline. We can see the comparison of quality between the two variants in Figure 5.8b. The plot clearly shows, that the difference in quality between the baseline and the variant with iteration limits is very small, there are only four instances, for which the result of the baseline is larger. As with the configuration for `AIDS` instances we thus clearly see, that the chosen configuration is not over fitted to the test data and applies to unseen instances as well.

Given these results, we apply the appropriate configurations for the experiments used in the overall comparison in Section 5.8. If no prior knowledge about the input data exists, it is difficult to estimate a suitable configuration. Therefore, we recommend to always perform experiments on a subset of instances in order to understand, which configurations may be suitable.

5.4.2 Branch-Reduce

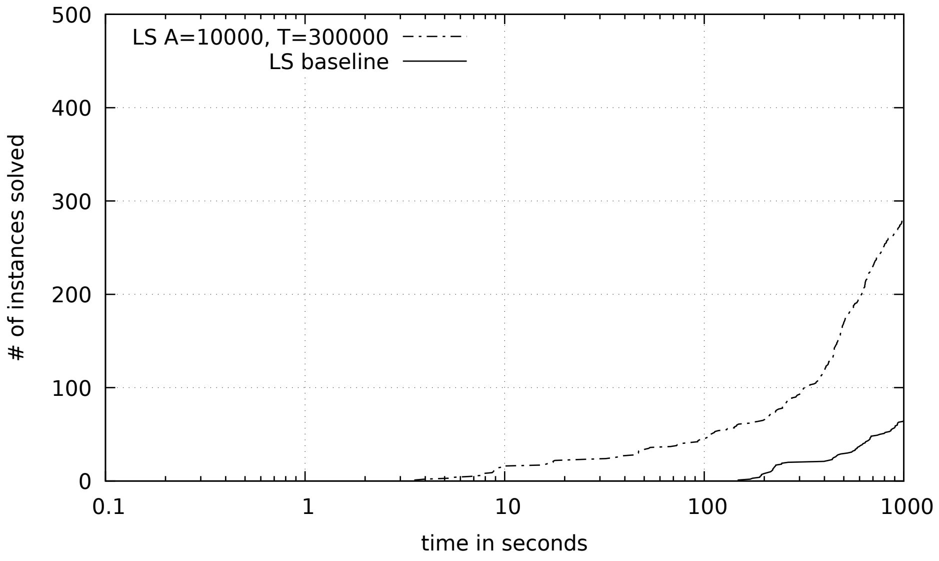
Next, we investigate the difference in running time of the Branch-And-Reduce solver when using our MCS specific reduction style, versus the default styles. For this experiment, we look at all instance groups.

We first investigate the difference in running time and result quality for `AIDS` instances. Looking at Figure 5.9, we can clearly observe that our MCS specific reduction style outperforms both default styles. Many more instances can be solved within one second or less with the new style compared to the default styles. We observe that for more difficult instances, the running times of the dense and our custom style converge. The sparse reduction style performs worst overall, it is significantly outperformed by the other two styles. This again confirms that the product graph complement for `AIDS` instances is mostly dense. If we consider the result quality, there is no difference between the quality of all three styles, even though with the sparse reduction style more instances time out.

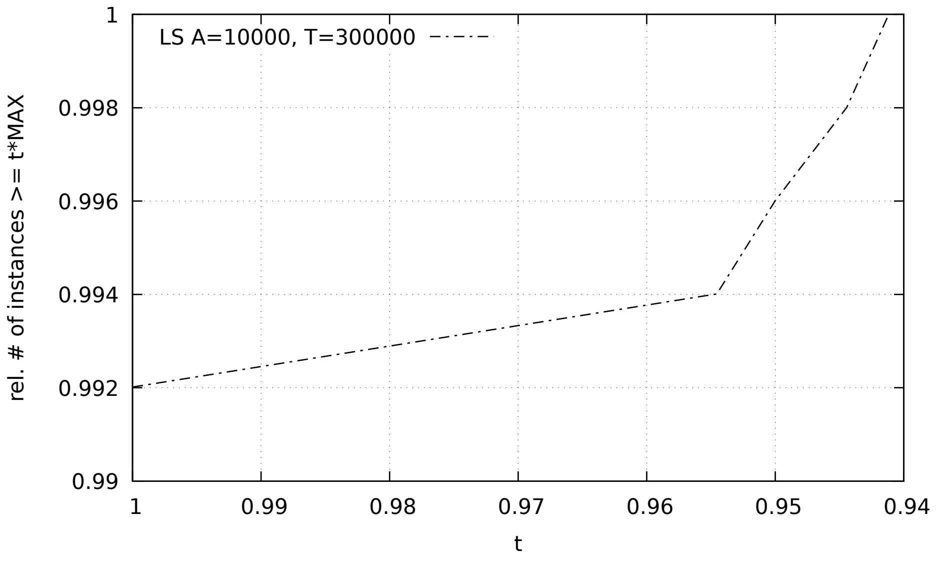
Next, looking at the `COX2` instances, we see in Figure 5.10a that our reduction style is again mostly faster than the two built-in styles. However, for this instance group the difference in running time is less pronounced. Interestingly, the sparse style is again outperformed by the dense style, and the running times of difficult instances for the dense and our custom style again converge. This suggests, that the dense reductions can be applied successfully more often than the other two reductions styles, even though the instances are comparatively sparse. And indeed, an analysis of the raw output data shows that the reductions of the dense style can be applied significantly more often than the reductions of the other two styles.

If we now look at the result quality for this instance set, we observe in Figure 5.10b a difference in quality between all three styles: The dense reduction style leads to overall slightly better quality than our MCS style, and the sparse style results in the worse quality. The dense style achieves a larger result on two additional and ten additional instances compared to the MCS and sparse styles, respectively.

For the `proteins` instances, we can see the running time comparison in Figure 5.11a. We observe that overall our custom style outperforms the other two styles significantly, far more instances can be solved within 10 seconds or less compared to the other styles. However, for more difficult instances, the performance of our custom style and the dense style are very similar, whilst the sparse style performs worse overall. In terms of result quality, we can see in Figure 5.11b, that our



(a) Instances solved over time



(b) Result quality

Figure 5.8: Comparison of running time and solution quality on previously unseen `mcsplain` instances for the chosen iteration limit configuration versus the baseline algorithm of Local Search.

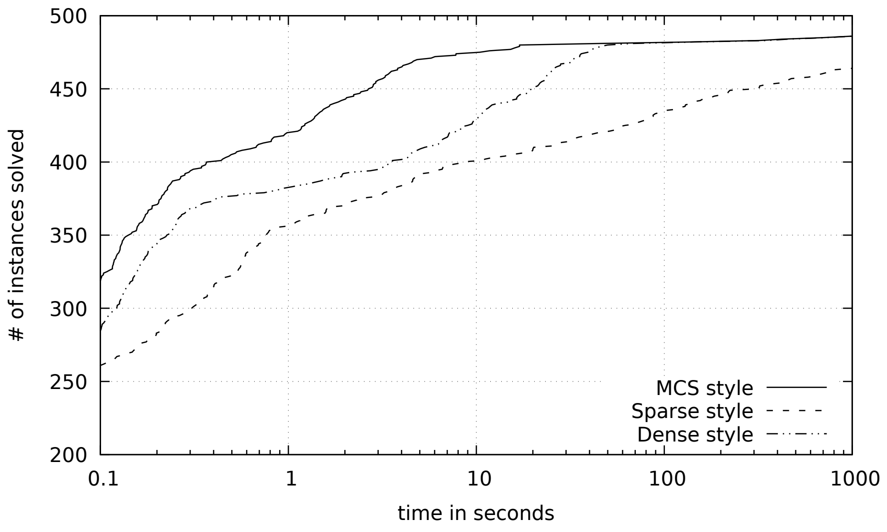


Figure 5.9: Instances solved over time on `AIDS` for the Branch-And-Reduce solver using different reduction styles.

custom style and the dense style achieve identical result quality, whilst the sparse style leads to slightly worse quality, there are three instances where the sparse style leads to a worse result than the other two styles.

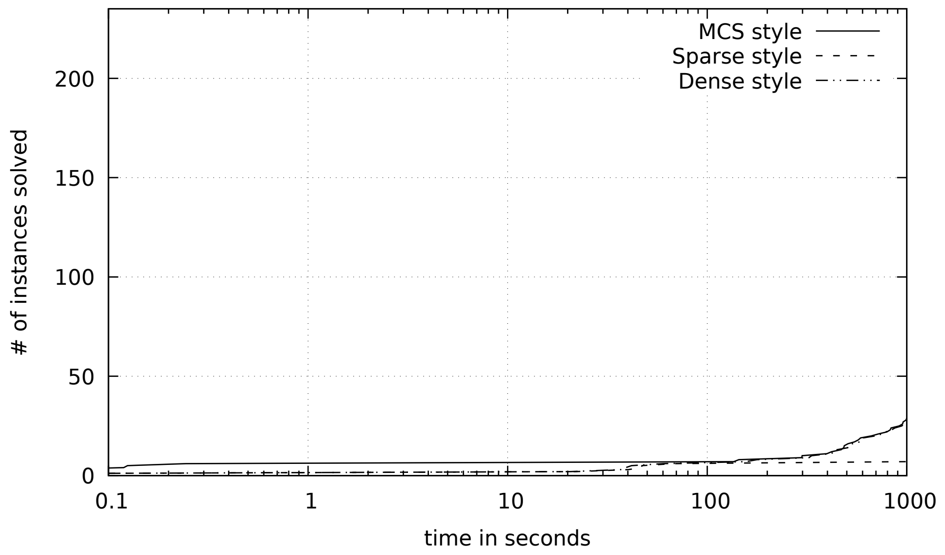
Next, we look at the results for `mcsplain` instances, where we can see the running time comparison in Figure 5.12a. We can clearly see, that again our custom style outperforms the two default styles. Additionally, here the sparse style performs significantly worse, requiring at least 400 seconds to solve an instance, whereas our custom style and the dense style can solve instances within 1 and 20 seconds, respectively. In terms of result quality, we again observe identical quality for our custom and the dense styles, as can be seen in Figure 5.12b. And again, the sparse style performs worse, here there are a total of 45 instances for which the sparse style finds worse results compared to the other two styles.

Finally, we compare the three styles on `mcsved` instances. Figure 5.13a shows the running time performance, where we for the first time observe almost identical running time performance for the dense style and our custom style. Again, however, the sparse style performs significantly worse. For this instance set, this is again surprising, as the instances are mostly sparse, therefore we would expect to see the sparse style outperforming the dense style. In terms of result quality, Figure 5.13b shows a slight difference between our custom and the dense style: There is a single instance, where the dense style achieves a larger result. Again, the sparse style performs worst than the other two styles, here it achieves a smaller result on four instances compared to the dense style.

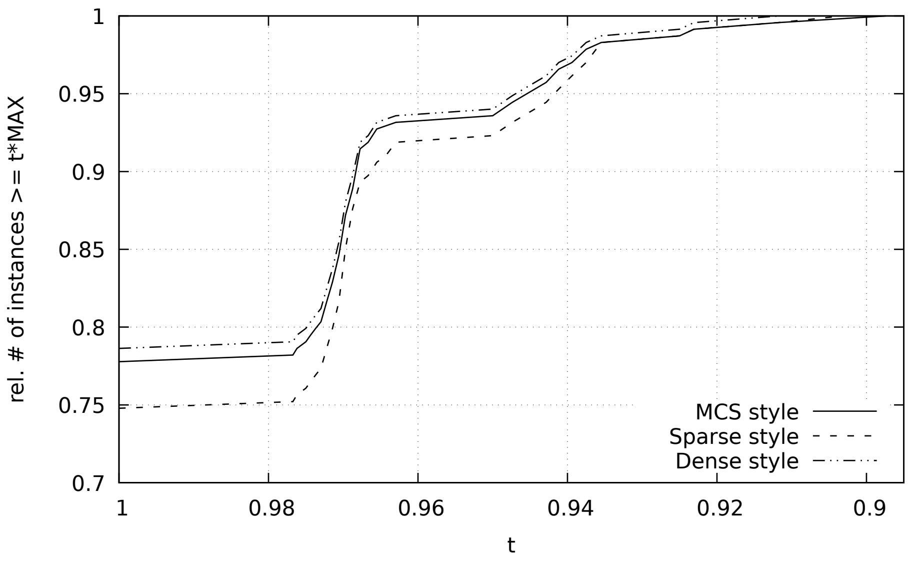
Overall, we consider the MCS reduction style to be superior, as the running time clearly improves, whilst the result quality at the worst case suffers slightly. We therefore from now on always activate our new reduction style for the Branch-And-Reduce as well as Local Search solvers of KaMIS.

5.5 Tight Bound and Domination Reduction

In this section we discuss the improvements made by the tight bound (Section 4.1) and the domination reduction (Section 4.3). For our experimental evaluation, we implemented both methods within KaMIS, therefore our discussion focuses on KaMIS. Note, that the domination

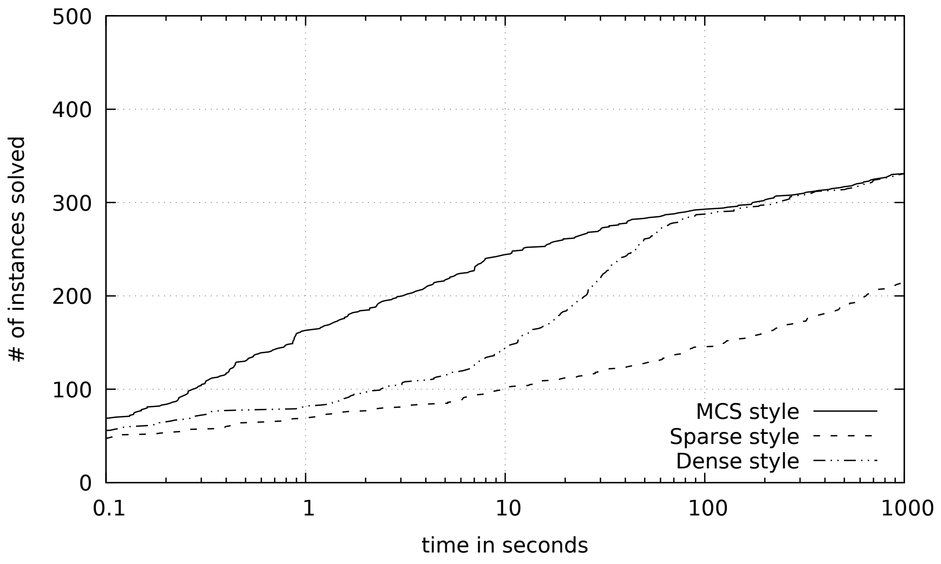


(a) Instances solved over time

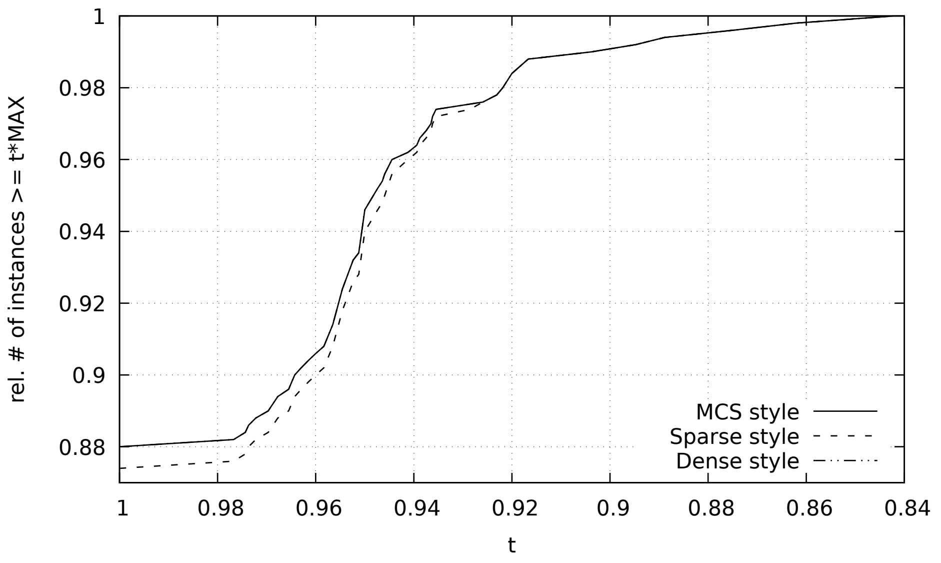


(b) Result quality

Figure 5.10: Comparison of running time and solution quality on `COX2` instances for the Branch-And-Reduce solver using different reduction styles.

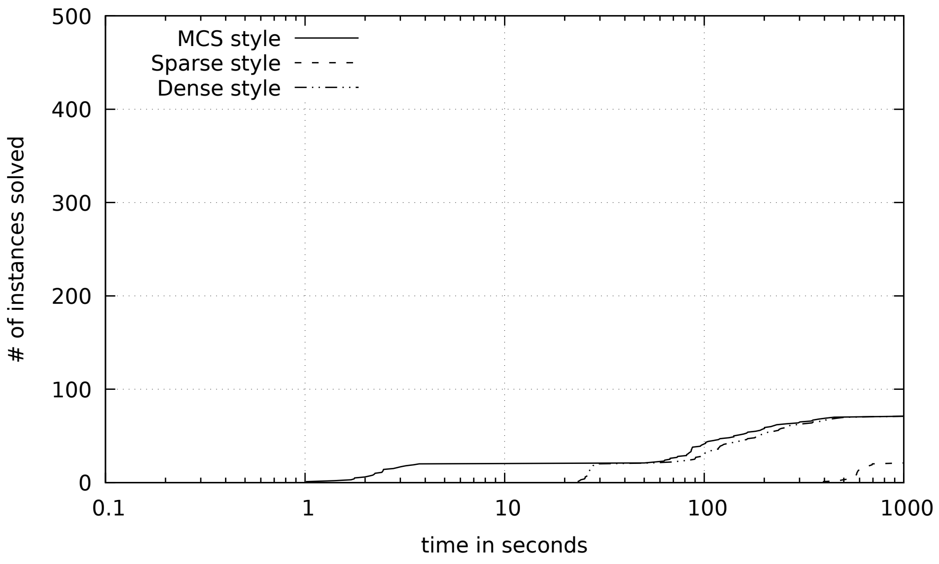


(a) Instances solved over time

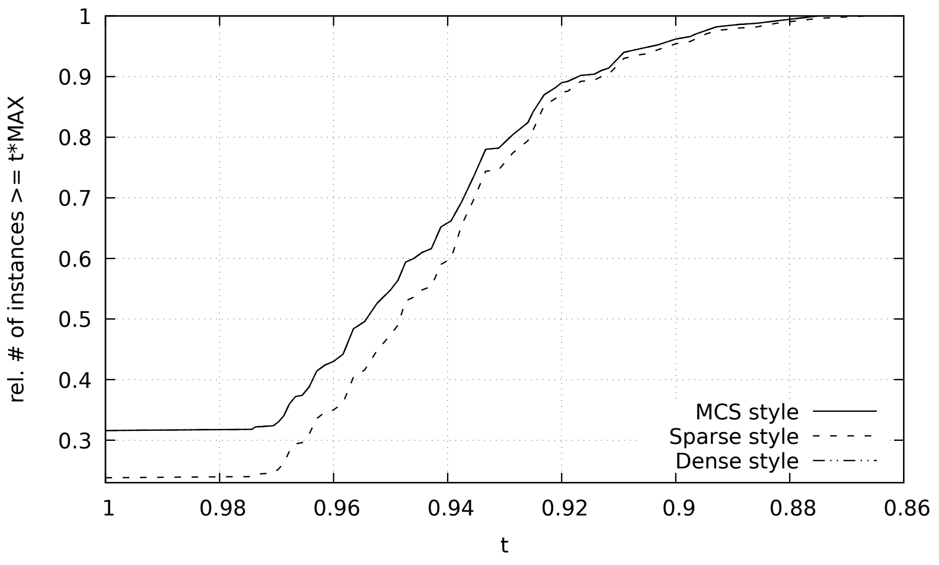


(b) Result quality

Figure 5.11: Comparison of running time and solution quality on `proteins` instances for the Branch-And-Reduce solver using different reduction styles.

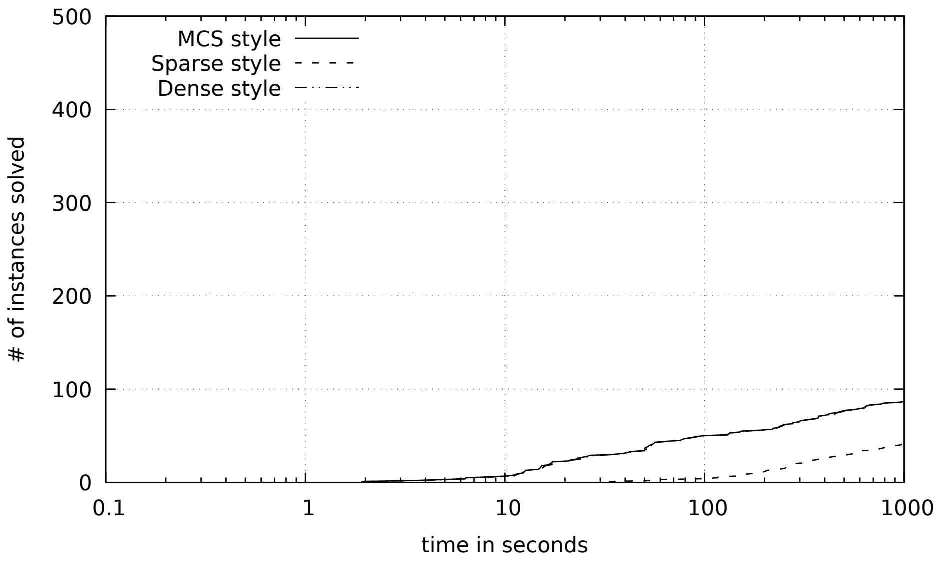


(a) Instances solved over time

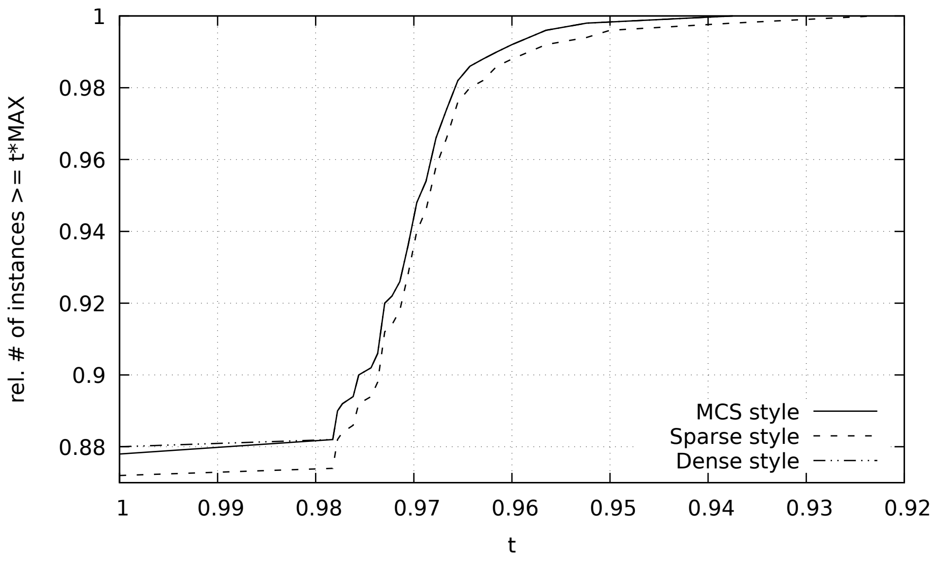


(b) Result quality

Figure 5.12: Comparison of running time and solution quality on `mcsplain` instances for the Branch-And-Reduce solver using different reduction styles.



(a) Instances solved over time



(b) Result quality

Figure 5.13: Comparison of running time and solution quality on `mcsved` instances for the Branch-And-Reduce solver using different reduction styles.

reduction could be applied to any solver, and the tight bound could be applied to any approach, which utilizes either the product graph or its complement. For the tight bound, we couple both methods described in Section 4.1 into one, as the independent set method should only be used together with a fallback method anyways. We additionally discuss the direct mapping described in Section 4.2 where applicable, as we implemented that for KaMIS as well and activated it per default. For this evaluation, we only experiment on the `AIDS` and `COX2` instances, as these present an ideal case for the upper bound and reductions, due to the many different vertex labels.

In Figure 5.14 we compare the number of instances solved over time for both `AIDS` and `COX2` instances when enabling either the tight bound or the domination reduction versus a baseline. For this experiment we chose these two instance groups, as they typically consist of many different labeled vertices, which is beneficial for both the domination reduction and the upper bound computation.

We can clearly observe, that both the tight bound and the domination reduction have little effect on the overall running time of the Branch-And-Reduce solver of KaMIS. Looking at the raw data, we see 68 domination reductions across 50 `AIDS` instances, and no reductions on the `COX2` instances. If we now look at how often the direct mapping is applied for the `AIDS` instances, we see 27 occurrences with the baseline version. However, with the domination reduction, the direct mapping occurs an additional 17 times, for a total of 44 invocation across all `AIDS` instances. For the `COX2` instances, the direct mapping never occurs at all.

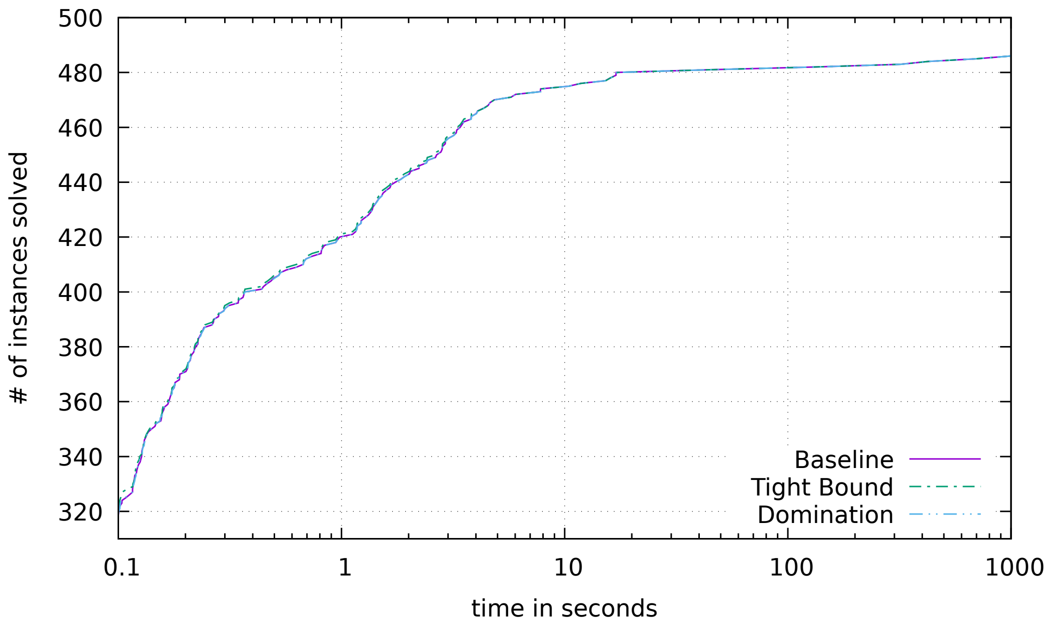
Regarding the upper bound, with `AIDS` we observe 41 instances, where the bound was reduced, out of those for 21 instances the bound was reduced by the independent set bound. For `COX2` instances, the bound was reduced on 31 instances, with 8 instances being reduced by the independent set. The updated upper bound was equal to the optimal solution on ten `AIDS` instances and no `COX2` instance. For the instances, where the new upper bound was equal to the optimum, the data suggests that these instances where fairly easy so solve anyways, which is why there is little difference in running time.

Even though neither the domination reduction nor the improvement upper bound yield a noticeable improvement in running time, we nonetheless activate both for all subsequent KaMIS experiments, as it does not affect the running time negatively.

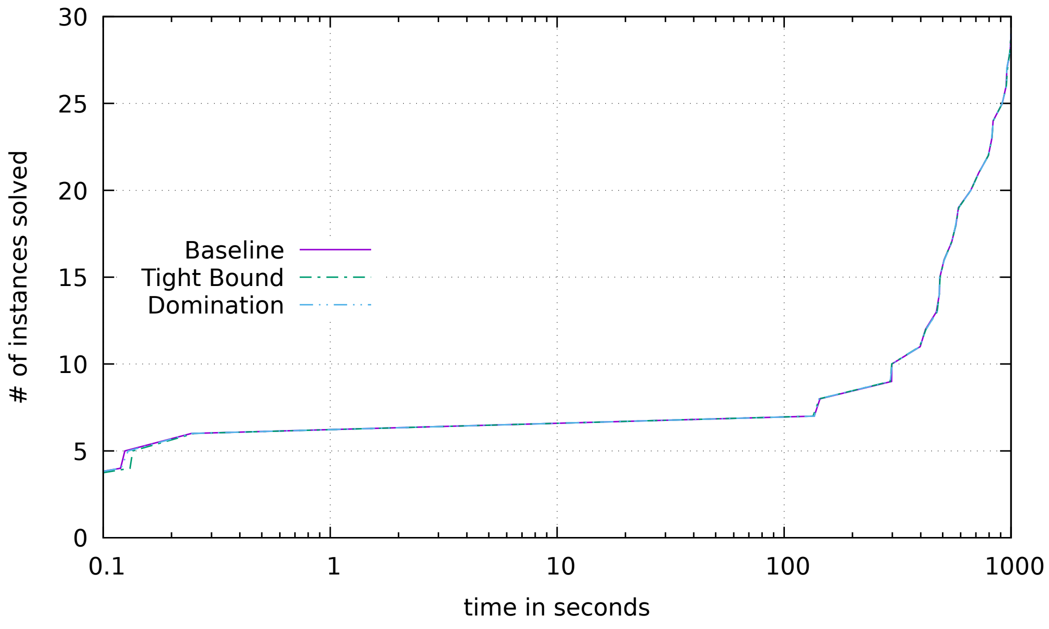
5.6 McSplit+RL Initial Scores

We now investigate the use of Weisfeiler-Leman and Jaccard-Index for initial scoring for McSplit+RL. Note, that the Weisfeiler-Leman scoring requires identical labels and thus identical neighborhoods. In preliminary experiments we discovered, that labels almost never match after the third iteration of Weisfeiler-Leman, we therefore limit this approach to three iterations. For the scoring based on the Jaccard-Index, the scores may increase for any arbitrary number of iterations. However, too many iterations are not useful either, as the additional over-head of computing these iterations increases and thus may overall slow down the execution. We therefore compare the results for $i \in [2..5]$ iterations. For our comparisons, we perform experiments on all instance groups.

First we take a look at the influence of the initial scores for the `AIDS` instance group. If we look at Figure 5.15a we see very little difference between the baseline McSplit+RL runs and the various initial scoring configurations. We notice three configurations, namely with Weisfeiler-Leman as well as 3 and 4 iterations of the Jaccard-Index, which are able to solve a single instance slightly faster than the baseline algorithm. However, the remainder of the running time is very close. If we now look at the result quality in Figure 5.15b, we observe a noticeable difference between the configurations: With 5 iterations of the Jaccard-Index, we observe the overall best quality, the Weisfeiler-Leman configuration and 3 iterations of the Jaccard-Index perform very similarly, albeit slightly worse than 5 iterations. With 2 and 4 iterations, we notice mostly superior quality to the baseline algorithm as well, except for one single instance, where the quality of the baseline algorithm is slightly superior.

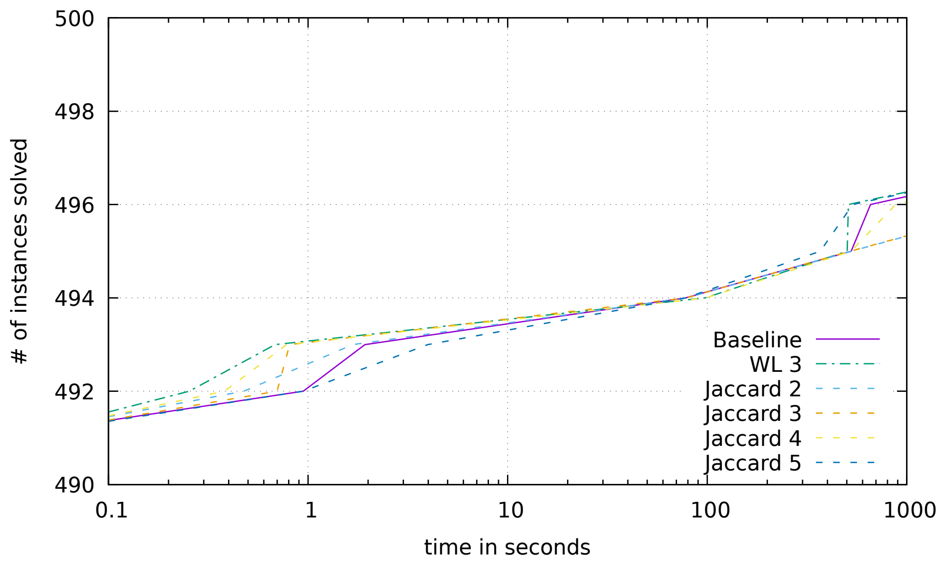


(a) `AIDS` instances

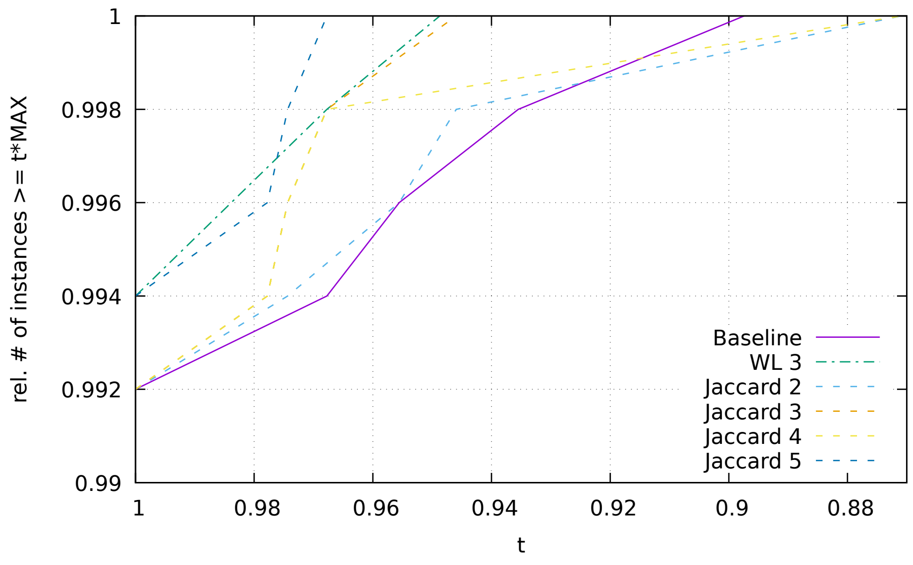


(b) `COX2` instances

Figure 5.14: Tight Bound and Domination Reduction on two instance groups

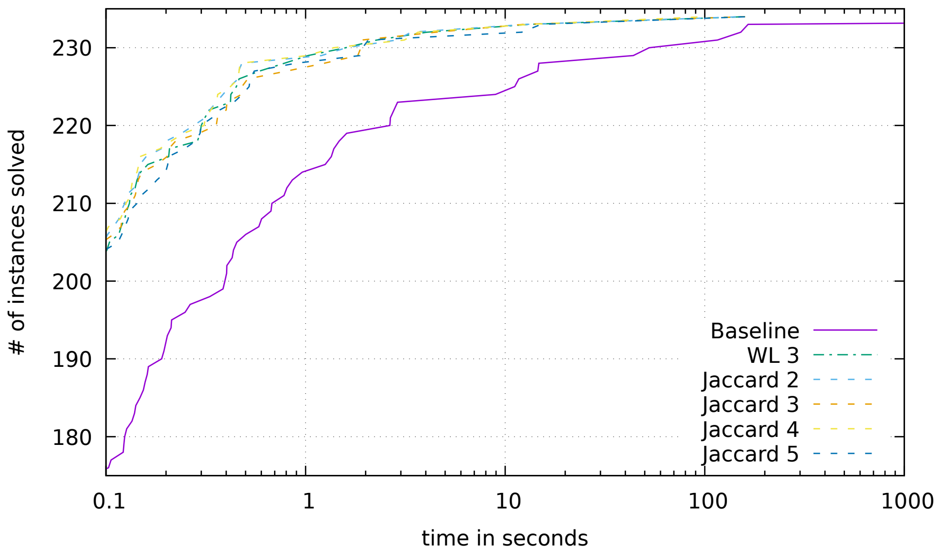


(a) Instances solved over time

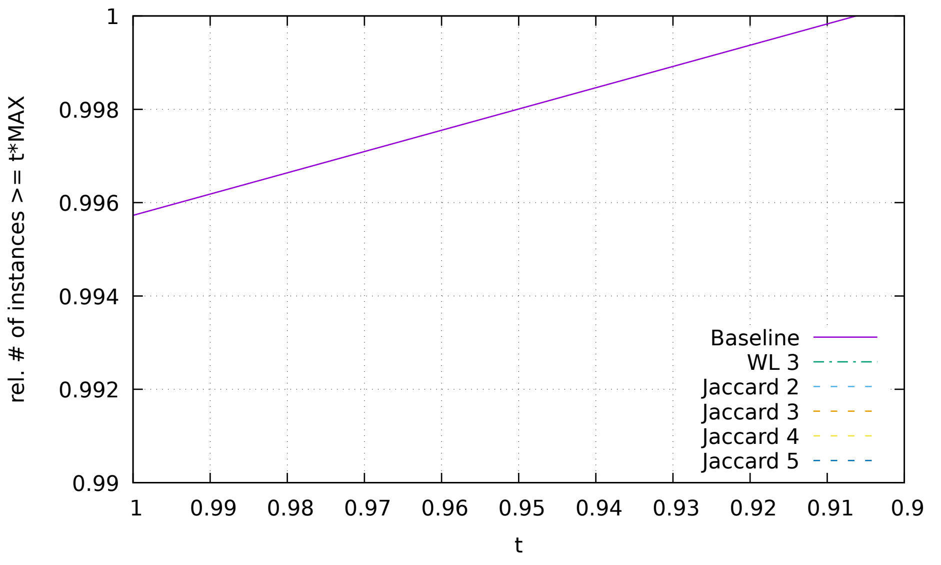


(b) Result quality

Figure 5.15: Comparison of running time and solution quality on `AIDS` instances for different initial scoring methods for McSplit+RL.



(a) Instances solved over time



(b) Result quality

Figure 5.16: Comparison of running time and solution quality on `COX2` instances for different initial scoring methods for McSplit+RL.

Next, we compare the configurations with the baseline algorithm on the COX2 instances. As we can see in Figure 5.16a, we observe a drastic difference between the various initial scoring configurations and the baseline algorithm. While the baseline algorithm times out on one instance, all initial scoring configurations finish all instances within roughly 110 seconds, significantly outperforming the baseline. More significantly, the baseline algorithm computes the optimum result for 176 instances within less than one millisecond, whereas all configurations with initial scores finish about 205 instances within that time frame. Comparing the various configurations among themselves, we notice very similar running times, with Jaccard-Index with 2 and 4 iterations slightly outperforming the other approaches. Here, we again see very similar running times for the Weisfeiler-Leman scores as well. Looking at the result quality, the configurations with initial scores never time out and thus find the optimal solution for all instances, whereas the baseline algorithm times out on one instance and does not find the optimum on that instance, which is reflected in Figure 5.16b.

If we consider the proteins instances, Figure 5.17a shows, that the baseline algorithm performs very similarly, overall slightly better even than the variants with initial scores. Though the differences in running times are quite small. However, looking at Figure 5.17b, we observe a noticeable difference in result quality: The baseline algorithm solves at least one more instance to the maximum overall than all initial score configurations. Overall, however, the configurations with 3 and 4 iterations of Jaccard-Index achieve the best quality.

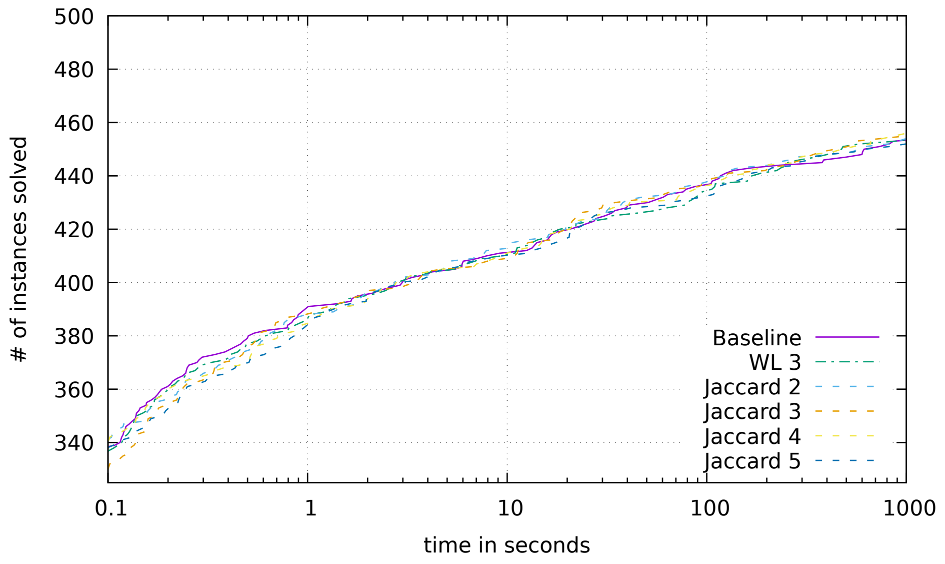
Next, we look at the mcsplain instances. Here, Figure 5.18a clearly shows that the baseline algorithm is consistently faster than any of our configurations, however, the difference in running time is again quite small. Interestingly, when looking at Figure 5.18b we see, that the baseline algorithm solves fewer instances to the maximum result any of our configurations. Overall, however, the baseline algorithm achieves the best worst-case result quality, with at the worst-case achieving at least 0.916 times the maximum value found. With 4 iterations of Jaccard-Index we achieve the second best worst-case quality, achieving at the worst-case 0.9 times the size of maximum result. If we look at absolute numbers, the baseline algorithm finds larger solutions on 24 instances than the configuration with 4 iterations of Jaccard-Index, however, the baseline algorithm finds smaller solutions on 39 instances. Interestingly, the different configurations seem to be beneficial on vastly different instances. If we directly compare the results with 4 and 5 iterations of Jaccard-Index, we find 27 instances where 4 iterations are better and 26 instances where 5 iterations are better.

Finally, we investigate the influence of the initial scores on the mcsved instance set. For this instance set, we observe almost no influence on the running times, as can be observed in Figure 5.19a. However, there is a slight improvement in result quality when using initial scores. Figure 5.19b shows, that the configuration with Weisfeiler-Leman as well as 2 iterations of Jaccard-Index overall perform slightly better than the baseline, whereas the other configurations initially perform slightly worse, yet overall lead to a slight improvement in quality.

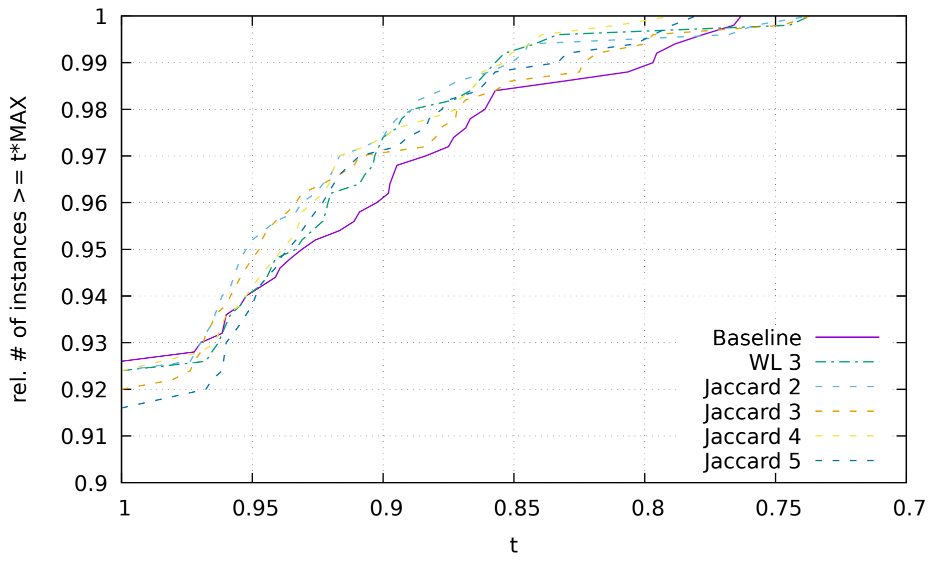
Our experiments indicate, that applying an initial scoring can improve the performance on most instance sets. We observe the most pronounced difference on the COX2 instance set, where the initial scoring helps outperform the baseline significantly. As all configurations have their ups and downs throughout the different instance sets, it is difficult to chose a single representative. However, with 4 iterations of Jaccard-Index we observe the most stable performance throughout, in terms of running time as well as result quality. We therefore from now on – unless otherwise specified – only use this configuration for the remainder of the experimental evaluation.

5.7 Exploiting Automorphisms

We next look at our method of exploiting automorphisms. Note, that we implemented the automorphism variants which ensure correctness only for McSplit+RL, though the results are likely to be similar for McSplit and KaMIS as well. We first discuss the automorphisms in the context of McSplit and McSplit+RL and next for the Branch-And-Reduce solver of KaMIS.

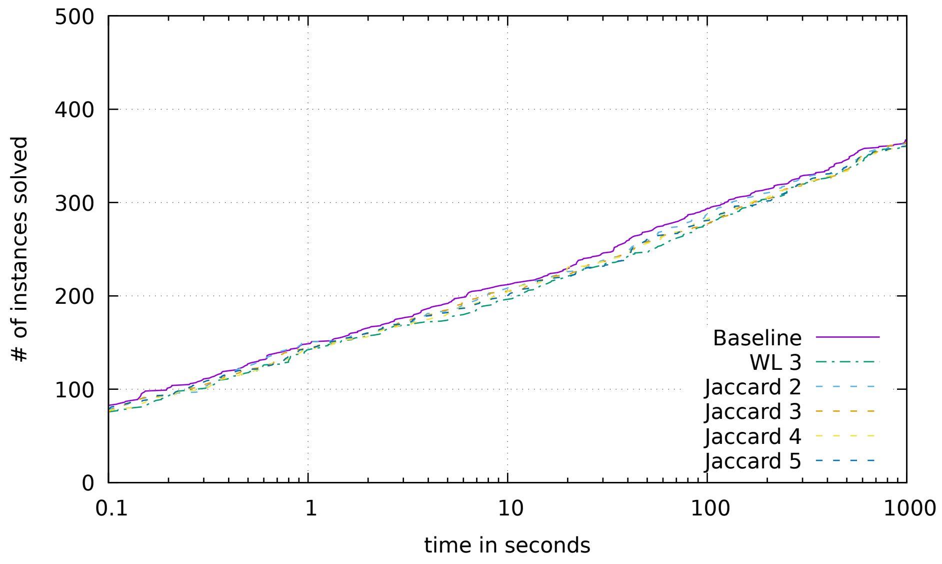


(a) Instances solved over time

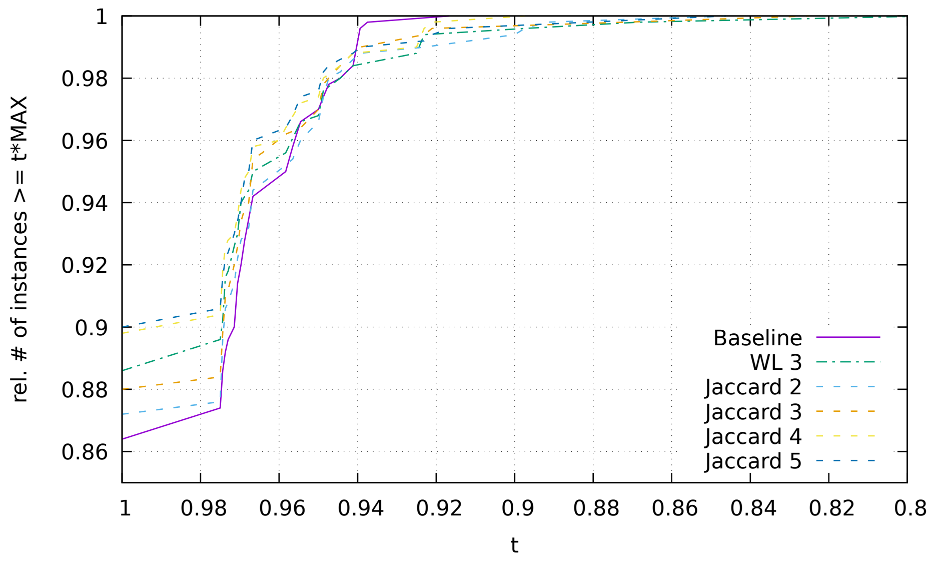


(b) Result quality

Figure 5.17: Comparison of running time and solution quality on `proteins` instances for different initial scoring methods for McSplit+RL.

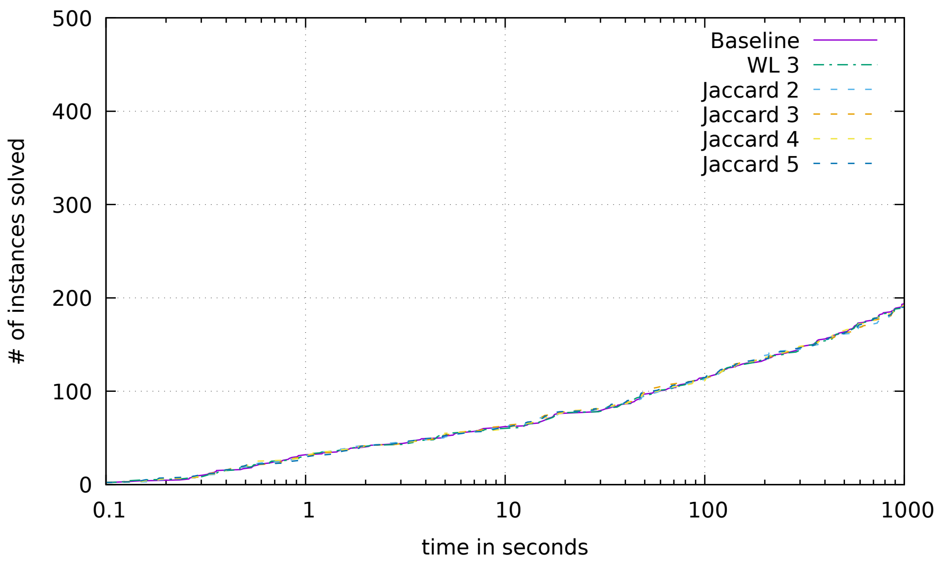


(a) Instances solved over time

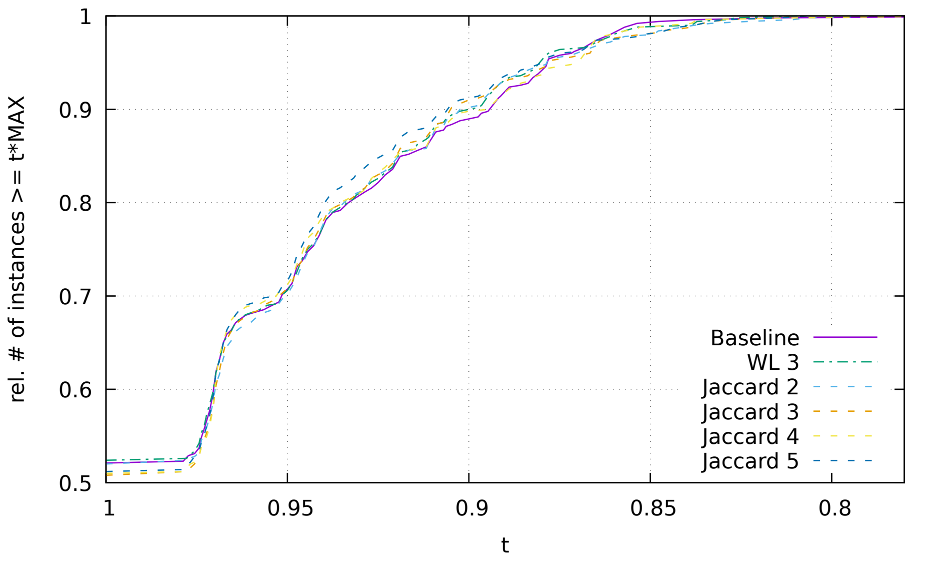


(b) Result quality

Figure 5.18: Comparison of running time and solution quality on `mcsplain` instances for different initial scoring methods for McSplit+RL.



(a) Instances solved over time



(b) Result quality

Figure 5.19: Comparison of running time and solution quality on `mcsved` instances for different initial scoring methods for McSplit+RL.

5.7.1 McSplit and McSplit+RL

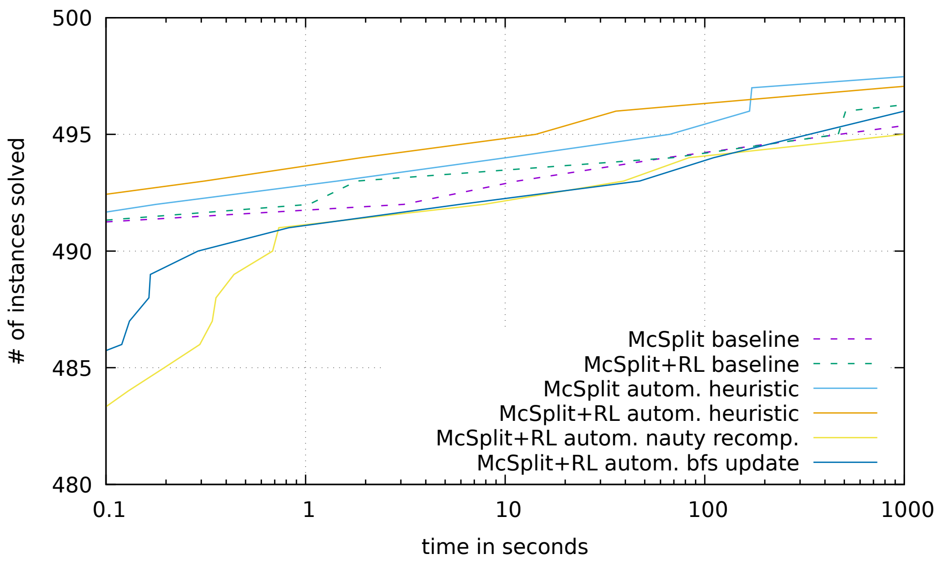
We first compare the baseline McSplit+RL and McSplit with the full automorphism recomputation (*nauty recomputation*), as well as our simple heuristic variant (*McSplit(+RL) autom. heuristic*) and the heuristic orbit update algorithm *BFS update*).

In Figure 5.20a we can see the running times of all approaches for `AIDS` instances. We first of all notice, that ensuring that automorphisms are exploited *correctly*, i.e. such that no optimal solution can be skipped, results in a significant slowdown when compared to the baseline algorithm. The BFS update algorithm leads to significant slowdowns as well, though not as drastic as the nauty recomputation. We additionally see, that the simple heuristic exploitation of automorphisms leads to better running times than the baseline algorithms, though the difference is rather small, only two or three fewer instances which result in a timeout. However, if we consider Figure 5.20b, we see a more drastic difference in result quality between the approaches. The variant, which recomputes automorphisms with `nauty` performs especially badly, leading to a significant decrease in result quality. Whereas the heuristic exploitation of automorphisms leads to a significant increase in result quality, even though the exploitation may potentially be non-optimal. For McSplit with heuristic automorphisms we see only one single instance, where the algorithm does not find the maximum overall solution, for McSplit+RL there are two such instances. If we compare the raw data, we see one single instance where the heuristic exploitation with McSplit+RL leads to a non-optimal result, that is not due to a timeout, whereas we observe no such occurrence with McSplit. The reason simply lies in the order in which branches are explored, which in the case of McSplit+RL with heuristic automorphisms and that single instance leads to a non-optimal result. Note, that a similar effect may easily occur vice-versa as well.

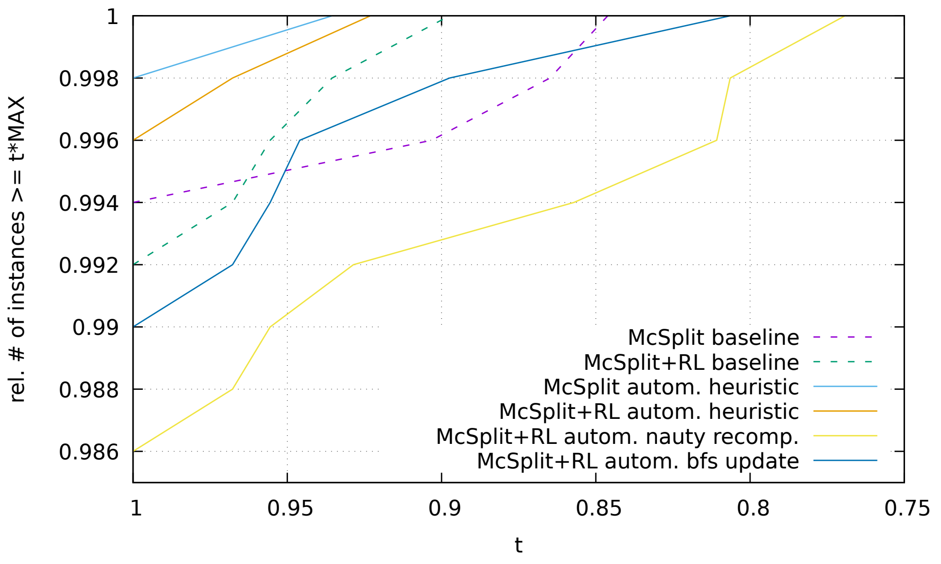
The difference in running time become more striking, if we look at the running times for `COX2` instances in Figure 5.21a. Remember, that the initial scoring for McSplit+RL already led to a significant increase of performance versus the baseline algorithm, such that all instances are solved within 110 seconds. With the heuristic automorphisms, McSplit+RL achieves another significant performance boost, solving all instances within less than 0.3 seconds. Similarly, McSplit with the heuristic automorphisms is able to solve all instances within less than six seconds and thus vastly outperforming its baseline. We again observe both the exact recomputation and the BFS update algorithm to perform worse than their baseline algorithm. In the case of the `COX2` instances, the difference between both approaches is more pronounced, leading to a significant decrease in performance for the full automorphism re-computation with `nauty`, whereas the BFS update only performs slightly worse than the baseline. Again, due to the heuristic nature of the automorphism exploitation we are not only interested in running time, but the result quality as well. We can see in Figure 5.21b, that the simple heuristic approaches for McSplit and McSplit+RL solve all instances to optimality, we again observe one instance with the baseline algorithm of McSplit+RL, which is solved slightly worse due to timeout. More strikingly, we see a significant quality decrease with the `nauty` re-computations, whereas with the BFS updates we find the optimal solution on all instances as well, regardless of the timeouts.

Given the poor performance of the orbit update routines, we disregard these two variants for the remainder of this section and only focus on the heuristic automorphism exploitation versus the baseline algorithms.

In Figure 5.22a we see that for `proteins` instances, the running time behavior is similar to that of `AIDS` instances. We again observe, that the automorphism heuristics outperform their respective baseline algorithms. For McSplit there are six and for McSplit+RL four fewer timeout instances with the automorphism heuristic enabled versus the respective baseline. Looking at the result quality in Figure 5.22b, we again observe the heuristic approaches to outperform their respective baseline algorithms. For McSplit the heuristic achieves larger results on ten instances, compared to the baseline, with McSplit+RL we observe 24 such instances, however, there are additionally ten instances where the McSplit+RL baseline achieves a larger result than the automorphism heuristic. Overall, for both algorithms the automorphism heuristic outperforms the baseline algorithm both in terms of result quality and running time.

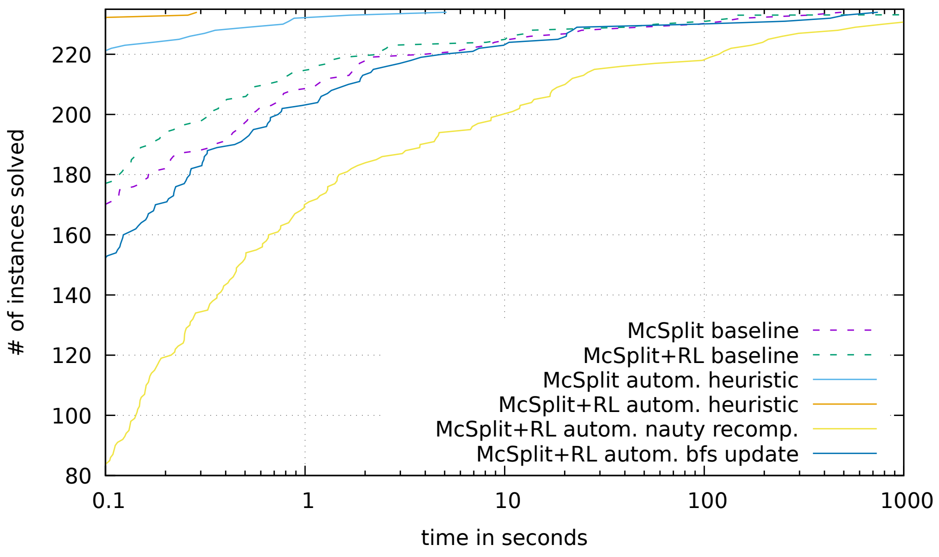


(a) Instances solved over time

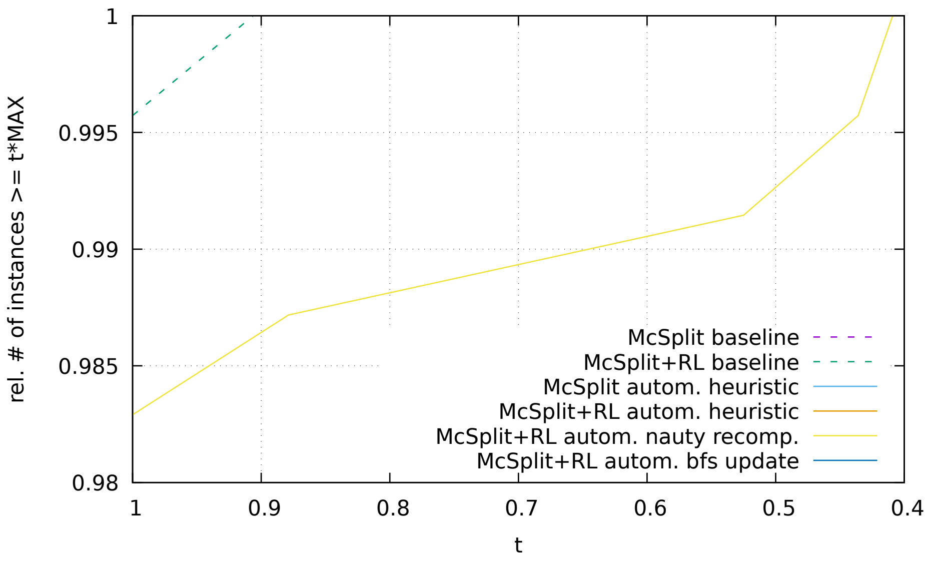


(b) Result quality

Figure 5.20: Comparison of running time and solution quality on `AIDS` instances for McSplit and McSplit+RL with heuristic automorphism exploitation *autom. heuristic* versus the correct automorphism exploitation (*autom. nauty recompute* and *autom. bfs update*) and the baseline of both algorithms.

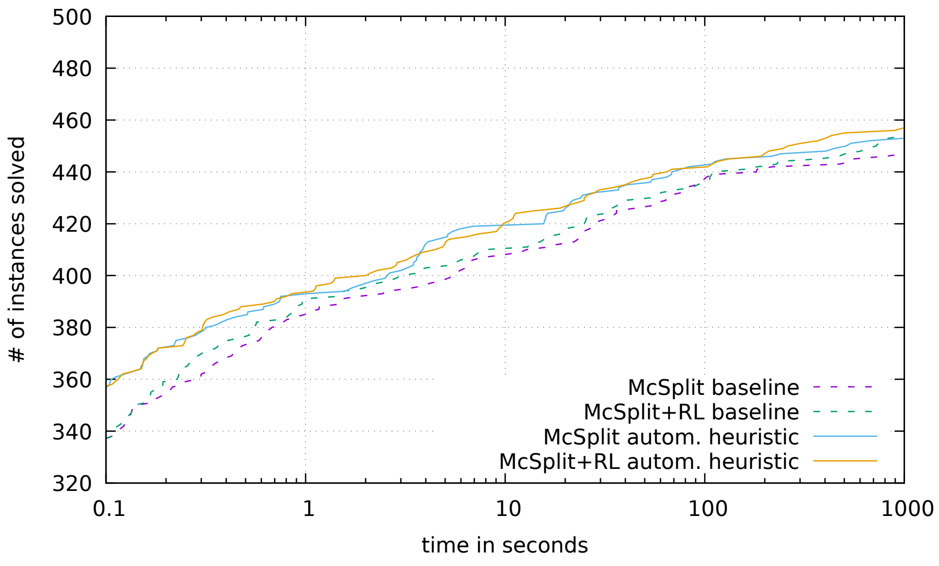


(a) Instances solved over time

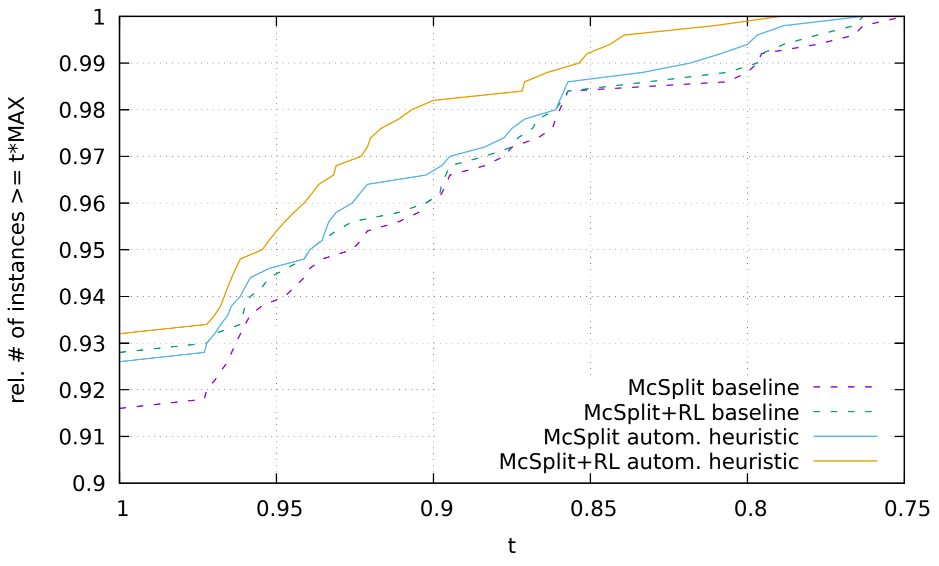


(b) Result quality

Figure 5.21: Comparison of running time and solution quality on `COX2` instances for McSplit and McSplit+RL with heuristic automorphism exploitation *autom. heuristic* versus the correct automorphism exploitation (*autom. nauty recompute* and *autom. bfs update*) and the baseline of both algorithms.

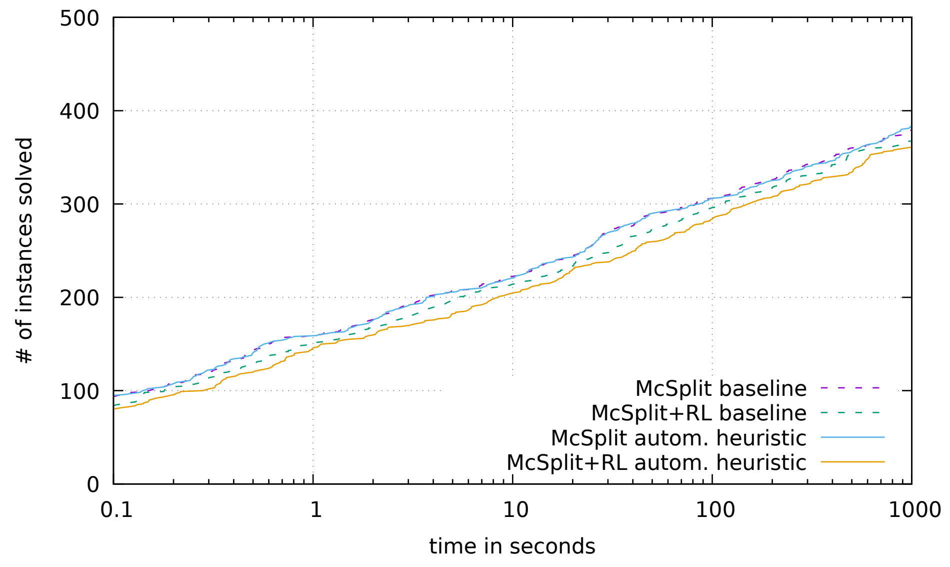


(a) Instances solved over time

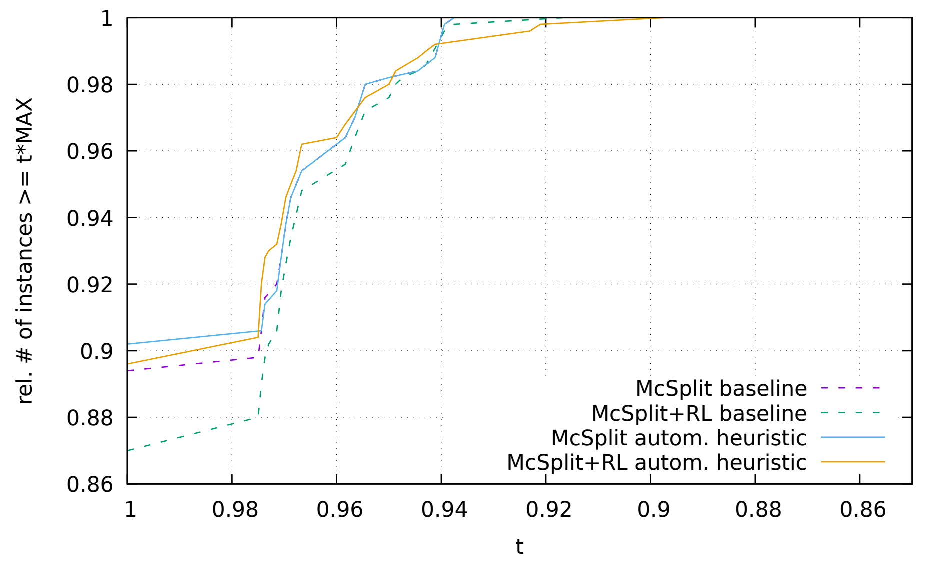


(b) Result quality

Figure 5.22: Comparison of running time and solution quality on `proteins` instances for McSplit and McSplit+RL with heuristic automorphism exploitation *autom. heuristic* versus the baseline of both algorithms.



(a) Instances solved over time



(b) Result quality

Figure 5.23: Comparison of running time and solution quality on `mcsplain` instances for McSplit and McSplit+RL with heuristic automorphism exploitation *autom. heuristic* versus the baseline of both algorithms.

Next, we look at `mcsplain` instances and discuss the running time behavior, depicted in Figure 5.23a. We, for the first time, observe that the automorphism heuristics do not improve the running time. This is not overly surprising, as the unlabeled `mcsplain` instances have few automorphisms and thus there is little exploitation to be done. We can observe, that McSplit performs very similarly with both the baseline as well as the automorphism heuristic, whereas for McSplit+RL we see a slight decline in running time performance for the heuristic. Nonetheless, if we look at Figure 5.23b, we do see an increase in result quality for the automorphism heuristics. For McSplit, the automorphism heuristic achieves a larger result on four instances compared to the baseline, whereas for McSplit+RL there are 21 such instances, thus indicating that the automorphism heuristics do improve the result quality, even though few automorphisms exist in the input graphs.

Finally, we turn our attention to `mcsved` instances. Here, Figure 5.24a shows, that the running time for both heuristics are very similar to their baseline variant, with little variations which may naturally occur during the experimentation. Looking at the result quality in Figure 5.24b, we see that for McSplit the baseline algorithm performs slightly better than the heuristic, there are eight instances, where the baseline achieves a larger result than the heuristic. For McSplit+RL, however, Figure 5.24b paints a slightly different picture: The result qualiyt of the heuristic is in total slightly worse than that of the baseline algorithm. For 59 instances, the baseline algorithm finds a larger solution than the heuristic, however, for 48 instances the heuristic finds a larger solution, which is a net difference of 11 instances for which the baseline algorithm outperforms the heuristic. Overall, the automorphism heuristics outperform the baseline algorithms in terms of running time performance as well as result quality. For the `mcsved` we do observe a slight drop in performance and quality, however, the result quality is still very close to the baseline algorithm. We thus utilize the automorphism heuristics for both algorithms in our overall evaluation of Section 5.8.

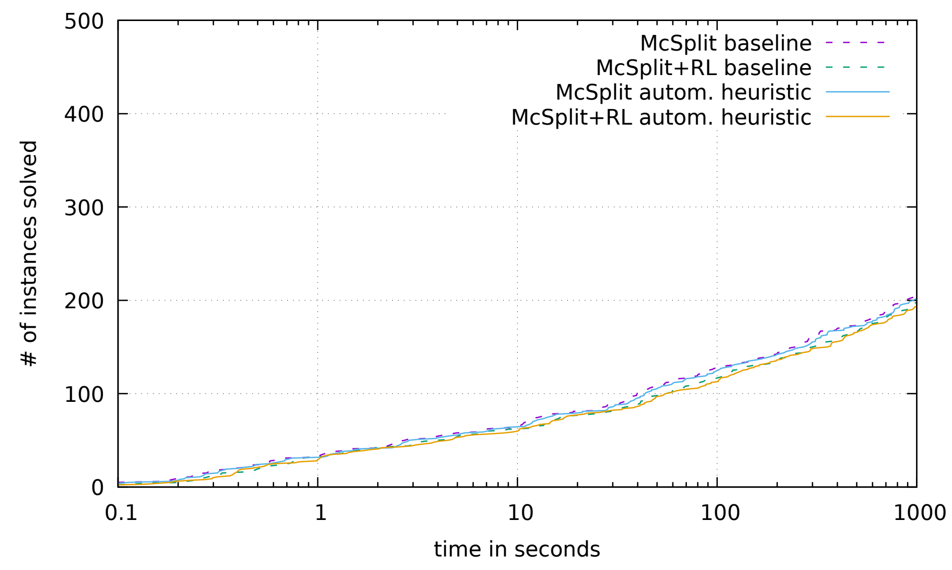
5.7.2 KaMIS

As we already saw with the automorphisms for McSplit+RL, ensuring correctness mostly leads to performance decreases, both in terms of result quality and running times. We therefore only consider the heuristic approach for the Branch-And-Reduce solver of KaMIS. Note, that we only consider automorphisms for the Branch-And-Reduce solver, but not the Local Search solver, as there is no clear cut way of exploiting automorphisms during the heuristic search. We again perform experiments on all instance groups.

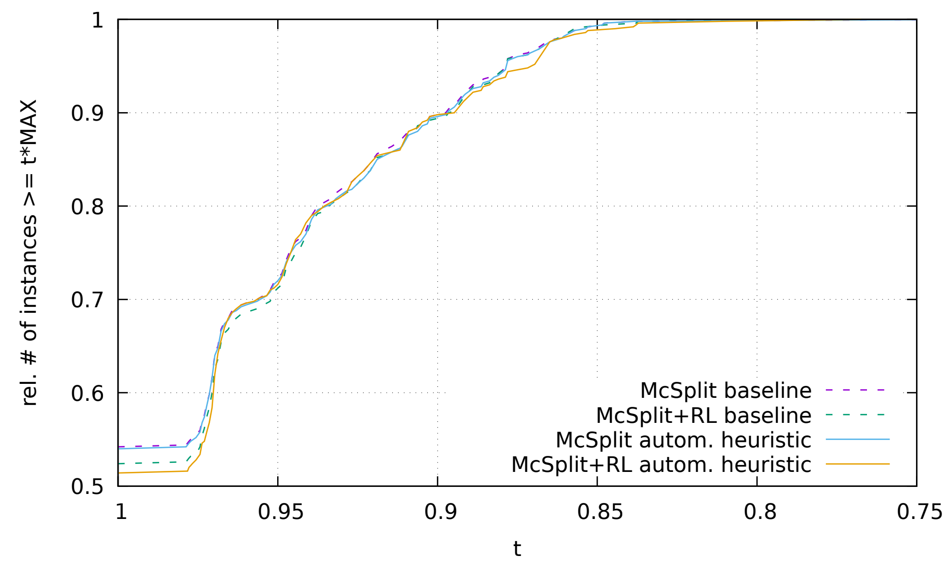
Figure 5.25a depicts the running times of the baseline Branch-And-Reduce solver versus the heuristic automorphism exploitation for `AIDS` instances. Note, that apart from the automorphisms both variants are configured identically with the best configurations discussed previously. We observe very little difference in the running time of the baseline and the automorphism variant, though with automorphisms the algorithm is slightly faster. Comparing the result quality in Figure 5.25b, we see absolutely no difference between both algorithms, as they achieve the exact same solution size for all instances.

However, if we compare the running times for `COX2` instances in Figure 5.26a, we observe an increase in solved instances for the configuration with automorphisms, though the difference is by far not as stark as it is with McSplit+RL. Looking at the difference in result quality in Figure 5.26b, we notice an improvement in quality with automorphisms enabled, as fewer instances time out. Even though the overall result quality is better, there are three non-timeout instances and three timeout instances, where the variant with automorphisms yields a smaller result than the baseline algorithm. However, there are 18 instances, where the heuristic achieves a larger result than the baseline algorithm.

When we look at Figure 5.27a, we again observe a boost in running time performance for our automorphism heuristic for the `proteins` instances. However, Figure 5.27b shows, that the faster running time comes at the cost of slightly reduced result quality. Overall, the baseline algorithm finds slightly larger solutions, there are two instances where the baseline algorithm finds smaller solutions than the heuristic, and five instances vice versa. Note, that the heuristic is able to solve

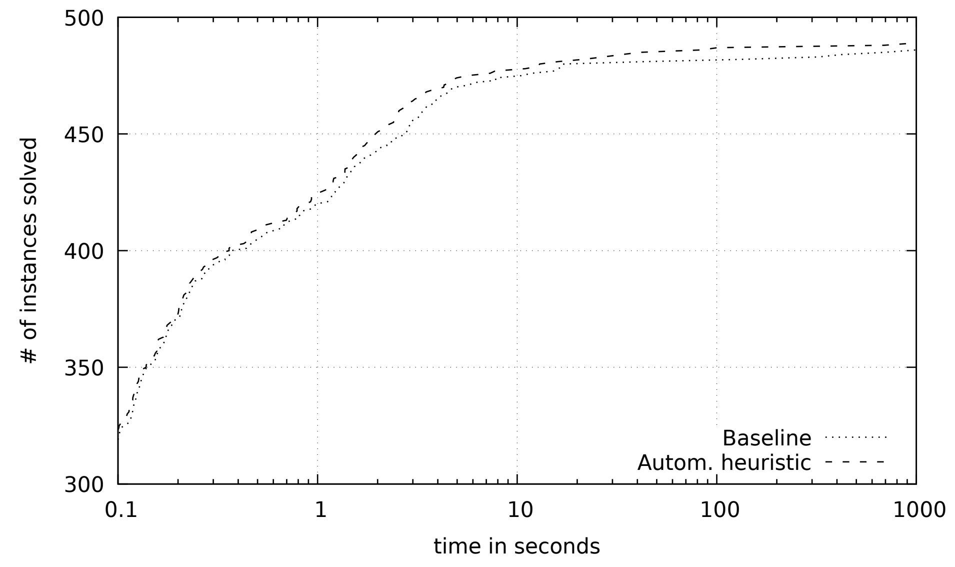


(a) Instances solved over time

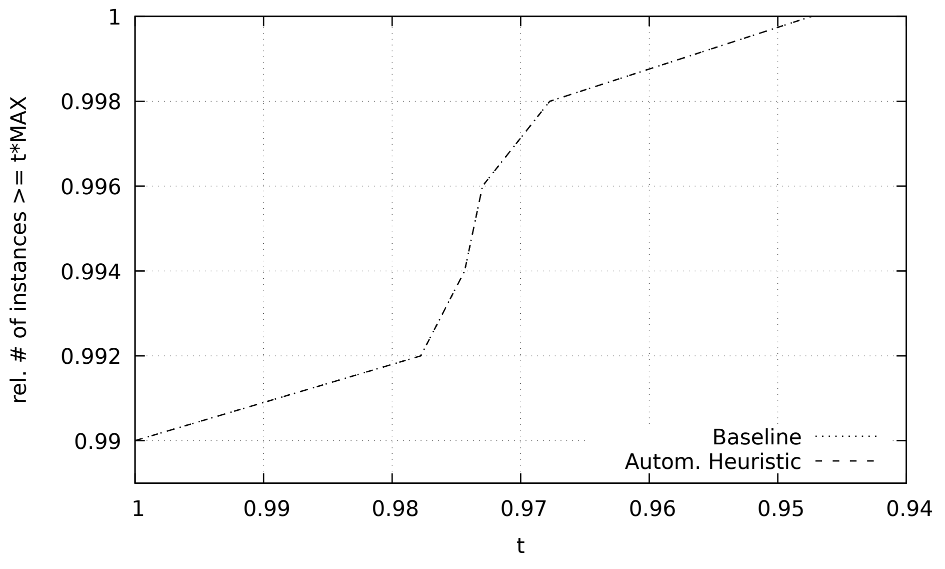


(b) Result quality

Figure 5.24: Comparison of running time and solution quality on `mcsved` instances for McSplit and McSplit+RL with heuristic automorphism exploitation *autom. heuristic* versus the baseline of both algorithms.

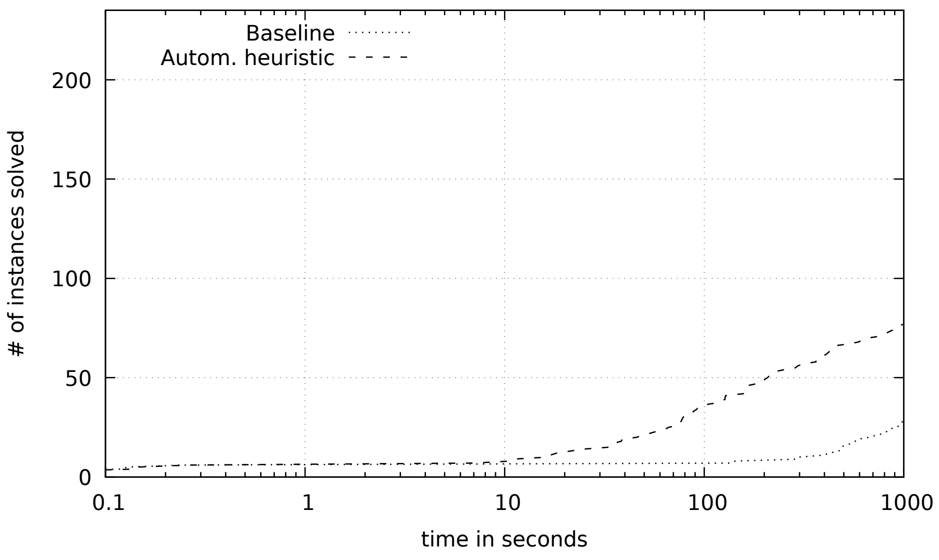


(a) Instances solved over time

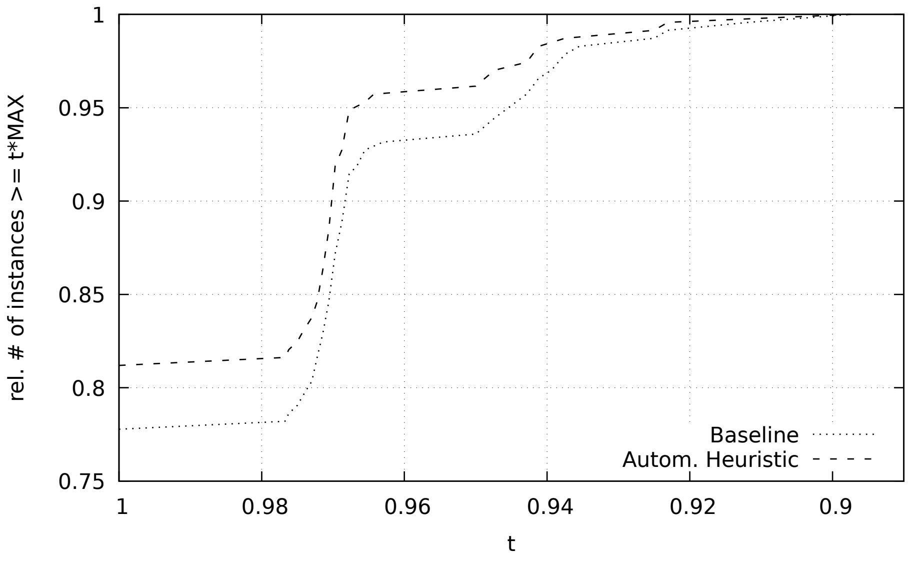


(b) Result quality

Figure 5.25: Comparison of running time and solution quality on `AIDS` instances for KaMIS with heuristic automorphism exploitation *autom. heuristic* versus the baseline algorithm.

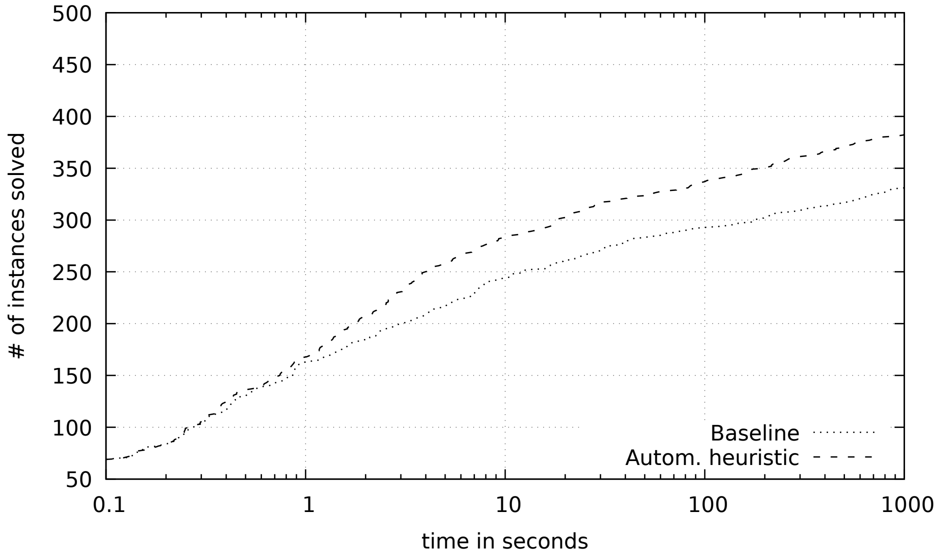


(a) Instances solved over time

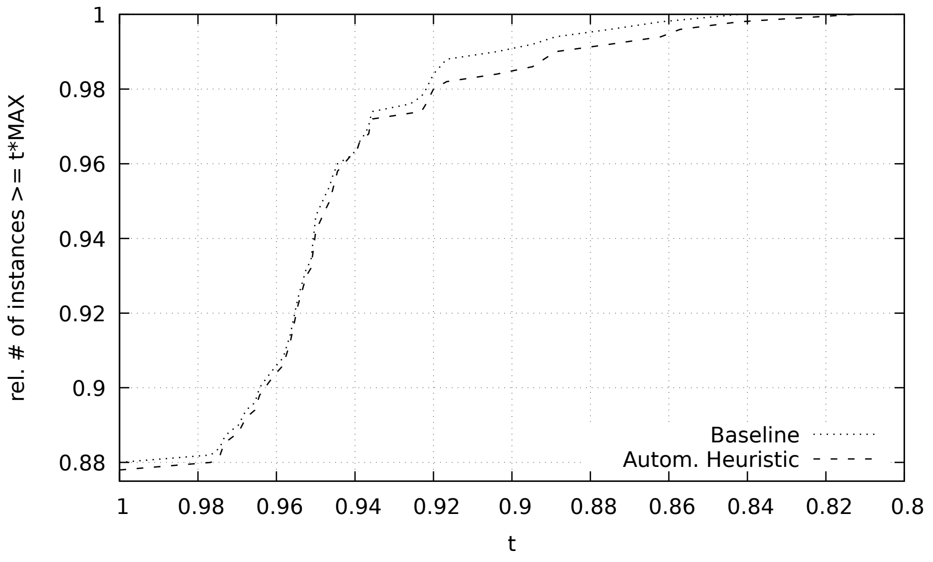


(b) Result quality

Figure 5.26: Comparison of running time and solution quality on `COX2` instances for KaMIS with heuristic automorphism exploitation *autom. heuristic* versus the baseline algorithm.



(a) Instances solved over time



(b) Result quality

Figure 5.27: Comparison of running time and solution quality on `proteins` instances for KaMIS with heuristic automorphism exploitation *autom. heuristic* versus the baseline algorithm.

<	Branch-Reduce	Local Search	McSplit	McSplit+RL	MoMC
Branch-Reduce	x	4	5	5	0
Local Search	2	x	3	2	2
McSplit	1	1	x	1	0
McSplit+RL	2	1	1	x	1
MoMC	5	6	6	6	x

Table 5.3: How many instances of the `AIDS` set was an algorithm able to solve with a larger solution than the other algorithms. Read: `ROW` achieved a smaller result on `X` instances compared to `COLUMN`.

roughly 50 instances more within the timeout than the baseline.

For `mcsplain` instances, we observe in Figure 5.28a a slight improvement in running time performance for the automorphism heuristic over the baseline algorithm. For these instances, the result quality between both variants is almost identical, as can be seen in Figure 5.28b. There are six instances, where the baseline achieves a smaller result than the heuristic, and five instances where the heuristic achieves a smaller result than the baseline.

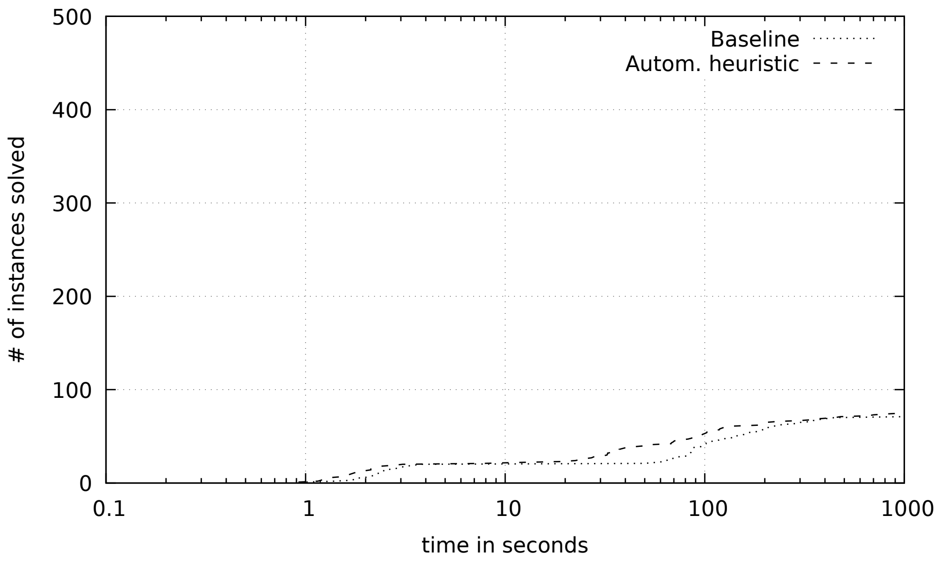
However, when we next look at `mcsved` instances, we can see in Figure 5.29a that both variants perform very similarly in terms of running time. Though when we consider the result quality we observe a significant drop in quality for the automorphism heuristic in Figure 5.29b. Overall, the heuristic finds the maximum result on 26 fewer instances than the baseline algorithm and the overall result quality is worse as well. More detailed, there are five instances which result in a smaller solution size for the baseline algorithm and a total of 31 instances, where the heuristic finds a solution smaller than the one found by the baseline.

Comparing all results, we observe an increase in performance with our automorphism heuristic for almost all our instance sets. For `proteins` instances we observe a small decrease in result quality but an increase in running time, only for `mcsved` instances do we observe significant performance drops w.r.t. the result quality. Overall, we consider the benefits of the heuristic to outweigh the downsides, we thus enable the automorphism heuristic for our overall comparison with the other approaches.

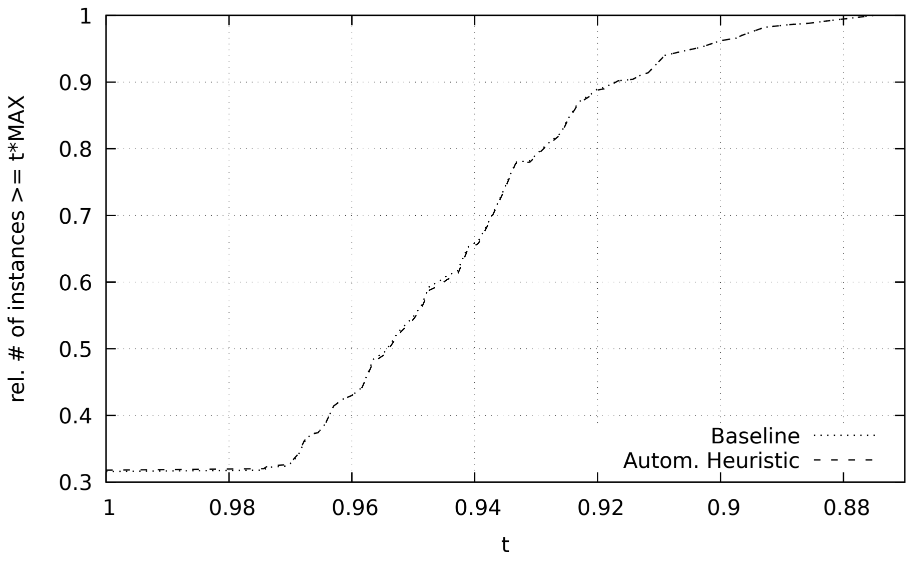
5.8 Overall Comparison

We now compare both solvers of KaMIS [LSS+19], McSplit [MPT17] as well as McSplit+RL [LLJH20] and the clique solver MoMC [LLJH20] with each other. Based on the preliminary experiments of the previous sections, we utilize the best configurations determined in our experiments. For the Branch-And-Reduce solver, we activate the tight bounds, the domination reduction as well as the automorphism heuristic. We additionally utilize the MCS reduction style for both the Branch-And-Reduce as well as Local Search solvers of KaMIS. For the Local Search solver we additionally utilize the iteration limits summarized in Table 5.2. Similarly to the Branch-And-Reduce solver, we activate the automorphism heuristic for both McSplit and McSplit+RL. The reinforcement learning of McSplit+RL is seeded with the initial scores of four iterations of our Jaccard-Index scoring method.

We again first discuss results regarding the `AIDS` instance set. The running times of the algorithms are given in Figure 5.30a. We can clearly see, that McSplit and McSplit+RL are vastly superior to the other approaches on this set, almost all instances can be solved by these to solvers within one millisecond. However, there are a few instances, where both solvers run into a timeout, whereas the Local Search runs in no timeouts at all, but fewer instances are solved within one millisecond. The clique solver MoMC performs quite well, it is able to solve close to 450 instances within one millisecond, and it only has a few more timeout instances than McSplit and

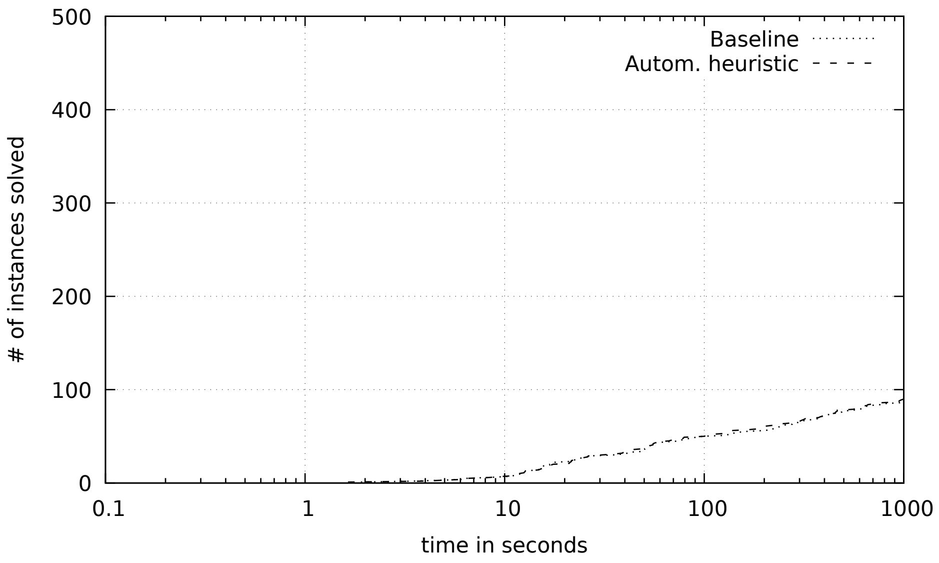


(a) Instances solved over time

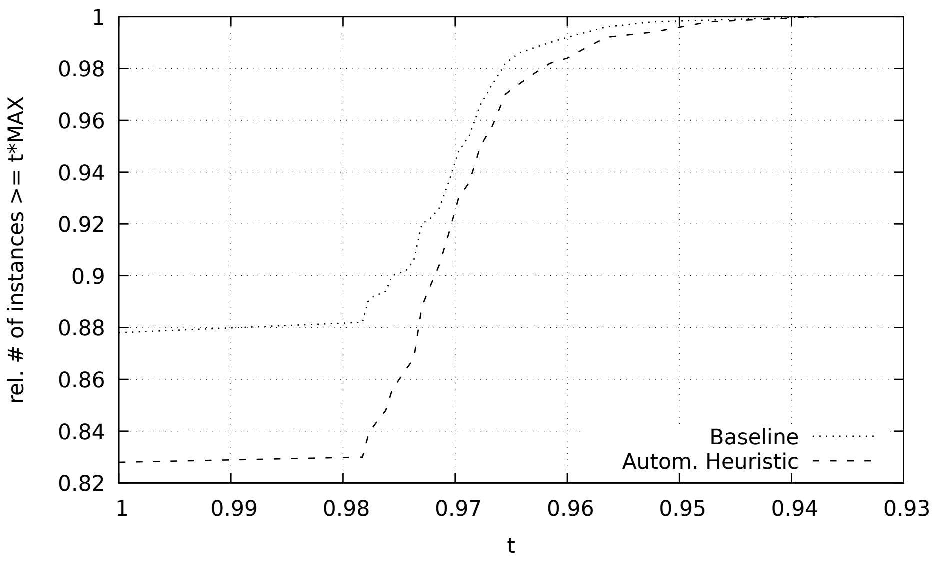


(b) Result quality

Figure 5.28: Comparison of running time and solution quality on `mcsplain` instances for KaMIS with heuristic automorphism exploitation *autom. heuristic* versus the baseline algorithm.

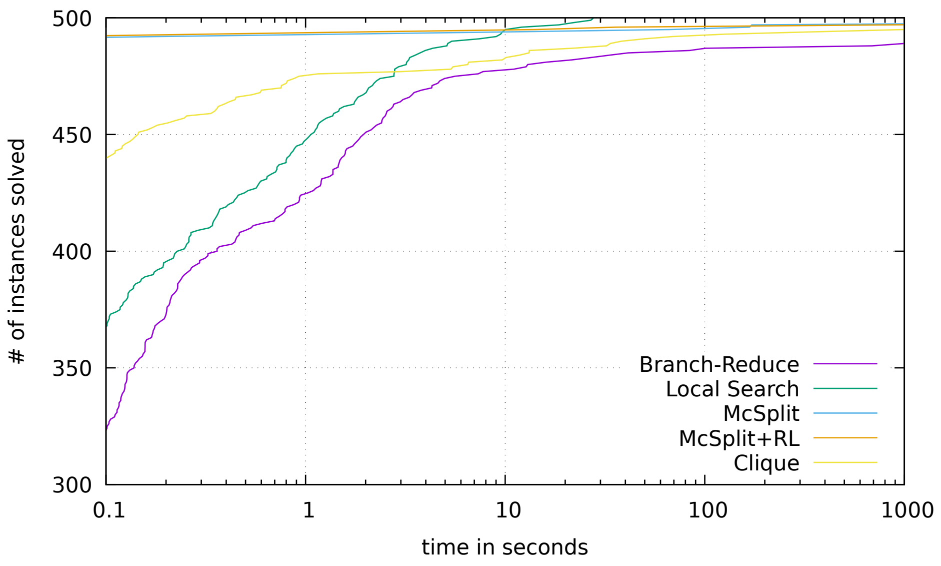


(a) Instances solved over time

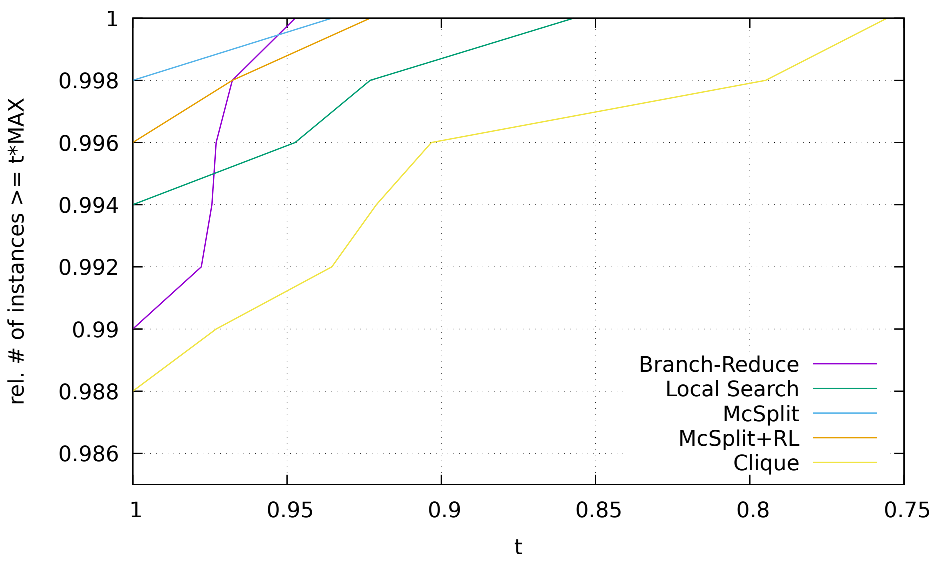


(b) Result quality

Figure 5.29: Comparison of running time and solution quality on `mcsved` instances for KaMIS with heuristic automorphism exploitation *autom. heuristic* versus the baseline algorithm.



(a) Instances solved over time



(b) Result quality

Figure 5.30: Comparison of running time and solution quality on `AIDS` instances for the best algorithms.

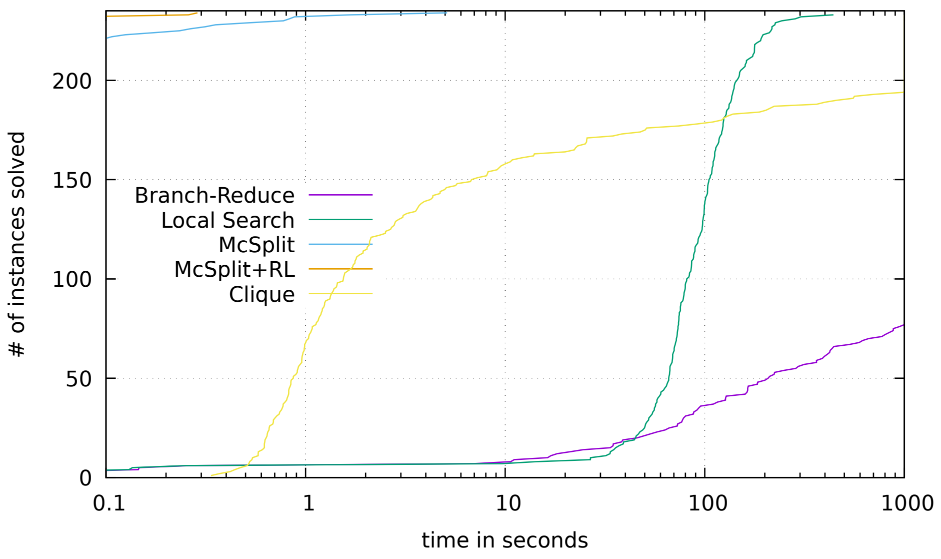
<	Branch-Reduce	Local Search	McSplit	McSplit+RL	MoMC
Branch-Reduce	x	28	44	44	32
Local Search	9	x	26	26	22
McSplit	0	0	x	0	0
McSplit+RL	0	0	0	x	0
MoMC	39	39	39	39	x

Table 5.4: How many instances of the `COX2` set was an algorithm able to solve with a larger solution than the other algorithms. Read: `ROW` achieved a smaller result on `X` instances compared to `COLUMN`.

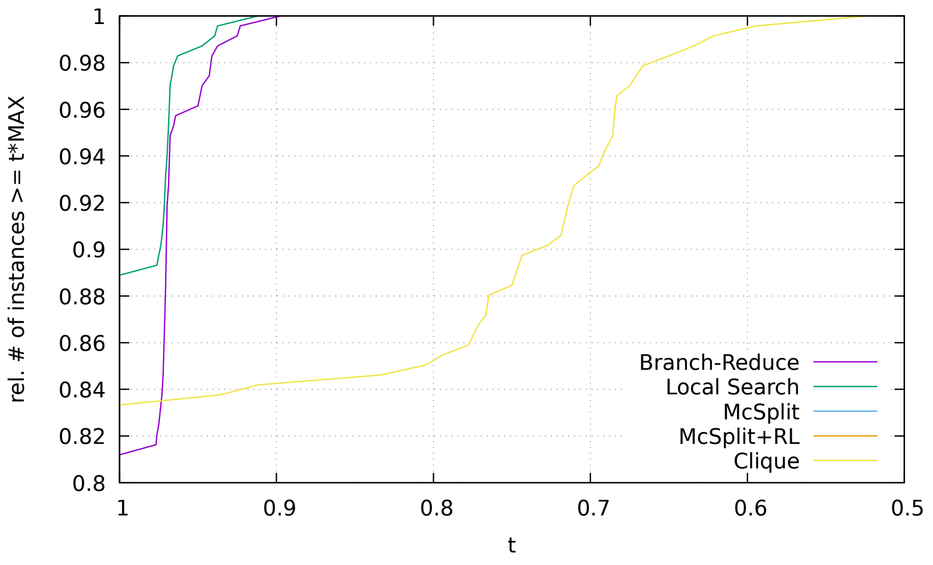
McSplit+RL. We clearly see, that the Branch-And-Reduce solver is the slowest overall, about 330 instances are solved within one milliseconds, and the number of instances with timeout is larger than for any other solver as well. Given these running time results, the result quality shown in Figure 5.30b is surprising: even though the Branch-And-Reduce solver was overall the slowest, in terms of result quality it is superior to MoMC and competitive to the other solvers, achieving at the worst-case almost 95% of the maximum solution size. The Local Search performs better than MoMC as well, solving all but three instances to the maximum result. Both McSplit+RL and McSplit perform very similarly and are overall best, however, McSplit has a slight advantage, with only one single instance not solved to the maximum, compared to two instances with McSplit+RL. With MoMC we observe the worst result quality, achieving the maximum result on only 494 and at the worst-case a quality of only 76% of the maximum solution found. Table 5.3 provides an alternative view on the result quality: The value X of a cell denotes, that the algorithm of the row achieved a smaller result on X instances compared to the algorithm of the column. This table clearly shows, that the Branch-And-Reduce solver found solutions at least as large as MoMC, regardless of the running time performance.

Next, for the `COX2` instance set, we see the running times in Figure 5.31a. We have already seen the outstanding performance of McSplit+RL for that instance set in Section 5.7, the running times achieved are clearly vastly superior in the overall performance as well. Second fastest is McSplit, while the other algorithms immensely struggle with this instance set. The Local Search is able to solve all instances without timeout, however, most instances require 50 seconds or more of running time. With MoMC we observe roughly 40 instances, which run into a timeout, but overall the performance is better than that of the Local Search, over 150 instances can be solved within ten seconds or less. And finally, the Branch-And-Reduce solver struggles the most with this instance set, only 80 instances do not result in a timeout. Again, looking at the result quality in Figure 5.31b is a bit surprising, as again the Branch-And-Reduce solver outperforms MoMC very clearly, the Branch-And-Reduce solver achieves at least 90% of the maximum result, whereas MoMC achieves at the worst-case only 55%. The Local Search again performs better than the Branch-And-Reduce solver, solving almost 20 more instances to the maximum. Here, Table 5.4 provides interesting insights: The Branch-And-Reduce solver achieves a smaller result than MoMC on 32 instances, whereas MoMC achieves a smaller result on 39 instances. Both McSplit and McSplit+RL achieve the best overall result on all instances, however, as we have already seen with the exact variant of McSplit+RL in Section 5.6, the solutions found by McSplit+RL are optimal, therefore no algorithm can perform better. Overall, the Branch-And-Reduce solver fails to find the optimum for 44 instances, whereas for MoMC there are only 39 such instances and for the Local Search solver there are 26 such instances.

Looking at the running times for `proteins` in Figure 5.32a, we again see that both McSplit and McSplit+RL are able to solve many instances within one millisecond, here it is roughly 350 instances. For both solvers, almost 50 instances run into the timeout. For MoMC, we observe 200 instances, for which the solver is able to find a solution within one millisecond and roughly 50 timeout instances as well. Next, the Local Search achieves the fewest instances solved within one

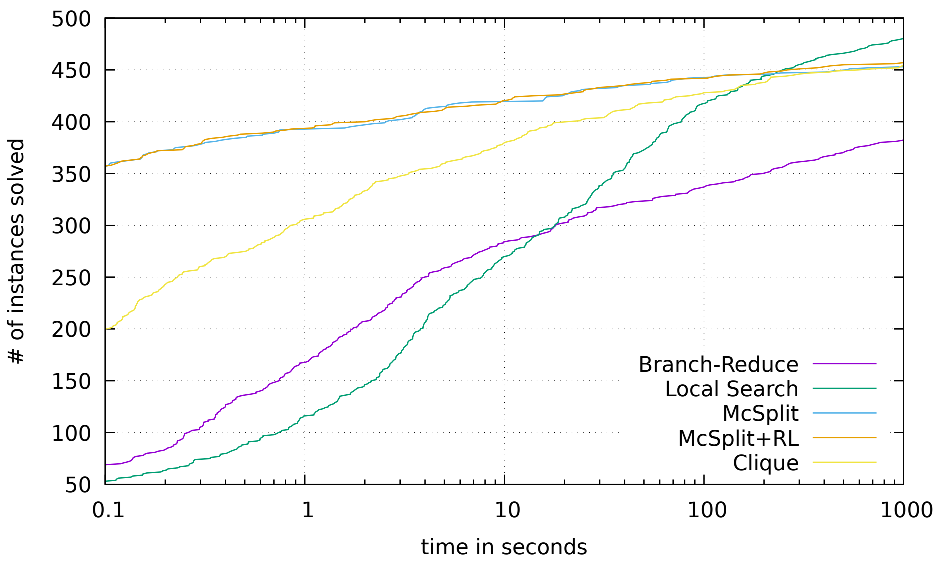


(a) Instances solved over time

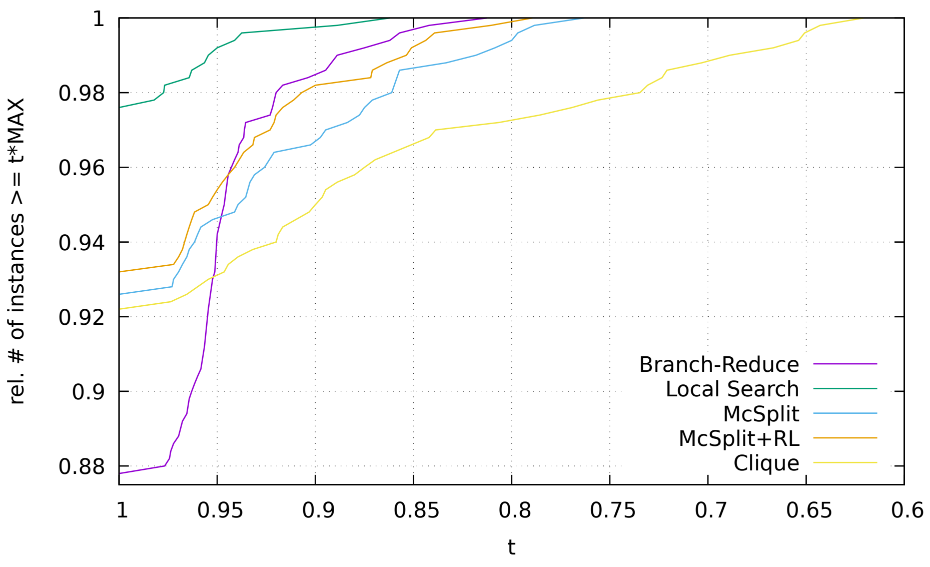


(b) Result quality

Figure 5.31: Comparison of running time and solution quality on `COX2` instances for the best algorithms.



(a) Instances solved over time



(b) Result quality

Figure 5.32: Comparison of running time and solution quality on `proteins` instances for the best algorithms.

<	Branch-Reduce	Local Search	McSplit	McSplit+RL	MoMC
Branch-Reduce	x	52	38	37	27
Local Search	0	x	7	7	6
McSplit	25	34	x	24	12
McSplit+RL	19	31	11	x	12
MoMC	29	39	32	31	x

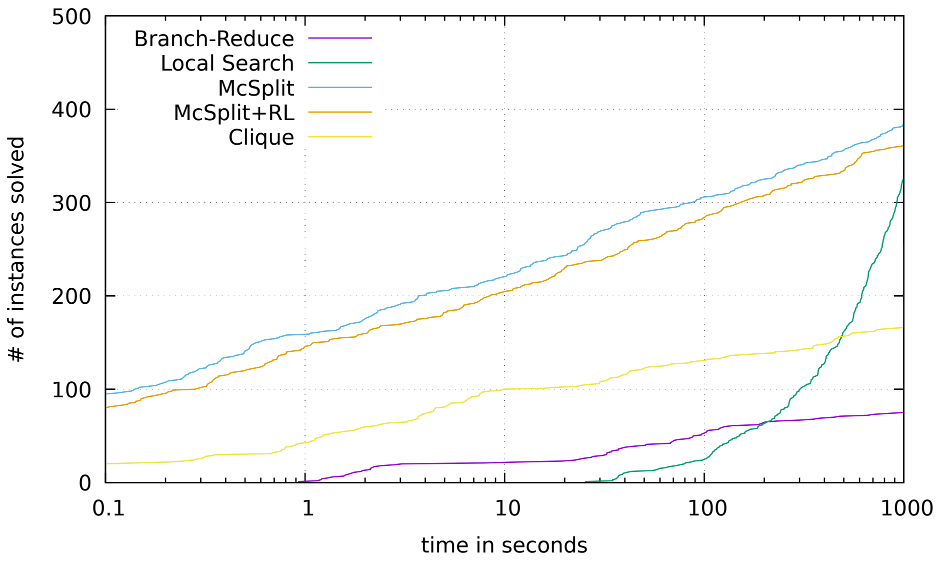
Table 5.5: How many instances of the `proteins` set was an algorithm able to solve with a larger solution than the other algorithms. Read: `ROW` achieved a smaller result on `X` instances compared to `COLUMN`.

<	Branch-Reduce	Local Search	McSplit	McSplit+RL	MoMC
Branch-Reduce	x	281	320	332	74
Local Search	1	x	145	141	21
McSplit	1	22	x	32	0
McSplit+RL	3	18	31	x	1
MoMC	245	281	295	292	x

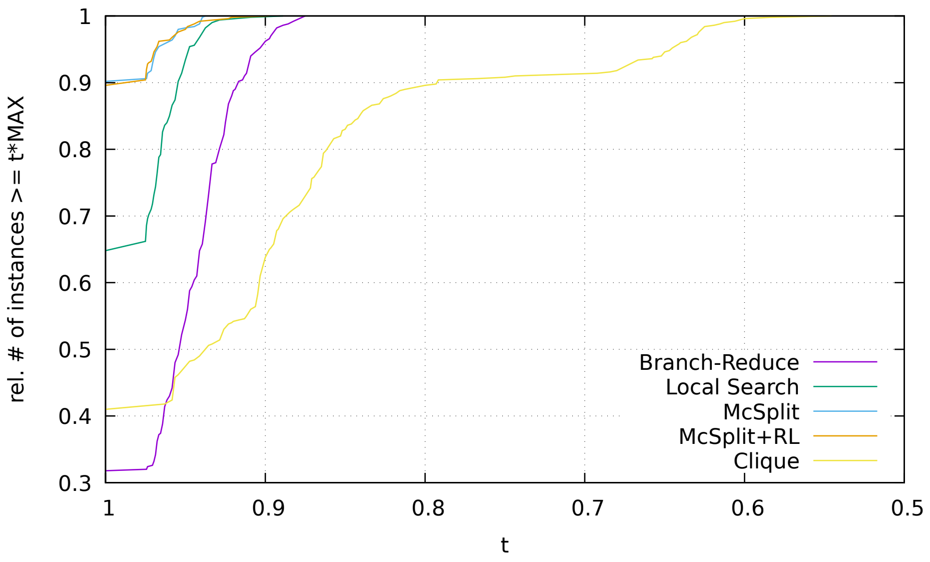
Table 5.6: How many instances of the `mcsplain` set was an algorithm able to solve with a larger solution than the other algorithms. Read: `ROW` achieved a smaller result on `X` instances compared to `COLUMN`.

millisecond, solving only 51 instances within that time, however, the algorithm has the fewest timeout instances, with only 20 timeouts. Lastly, the Branch-And-Reduce solver is able to solve a few more instances within one millisecond than the Local Search, however, overall the solver performs the worst and runs into timeouts for over 100 instances. As with the previous two instance sets, the running time plot is not a good indicator for what the result quality will be. In Figure 5.32b, we can clearly see, that the Local Search dominates the result quality, achieving the best result on almost 98% of instances, which translates to 12 instances in total, for which another algorithm achieved a better result. Next is McSplit+RL, which finds the maximum solution for all but 34 instances. Both McSplit and McSplit+RL perform very similarly, though McSplit+RL has a slight edge, finding consistently slightly better solutions. The clique solver MoMC finds the maximum solution on more instances than the Branch-And-Reduce solver, however, again, there are a few instances, for which MoMC finds comparatively bad solutions and as such, the quality can at the worst-case drop down to only 62% of the maximum solution. We again see a comparison of the solution sizes in Table 5.5. We can clearly see, that the Local Search performs best, there are 7 instances each, where McSplit and McSplit+RL find larger solutions, and 6 instances where MoMC finds a larger solution. However, the solutions found by the Local Search are larger than those of the other algorithms on many more instances.

We now compare the performance algorithms on the `mcsplain` instance set, where we can see the running time behaviors in Figure 5.33a. Here, we again observe dominant running times by McSplit and McSplit+RL, with McSplit performing slightly better of the two. McSplit is able to solve almost 100 instances within one millisecond, for roughly 110 instances McSplit runs into a timeout. Regarding the other algorithms, MoMC is able to solve a few instances within one millisecond and over 320 instances run into a timeout. Both the Branch-And-Reduce and Local Search solvers struggle with this instance set, the Branch-And-Reduce solver takes at least one second to solve an instance, the Local Search takes at least 25 seconds. It is therefore not surprising that the Branch-And-Reduce solver is able to solve less than 100 instances within the timeout, for the Local Search, however, we see a steep rise in instances solved, solving over 300 instances within the timeout. Again, looking at the result quality in Figure 5.33b, we see, that



(a) Instances solved over time



(b) Result quality

Figure 5.33: Comparison of running time and solution quality on `mcsplain` instances for the best algorithms.

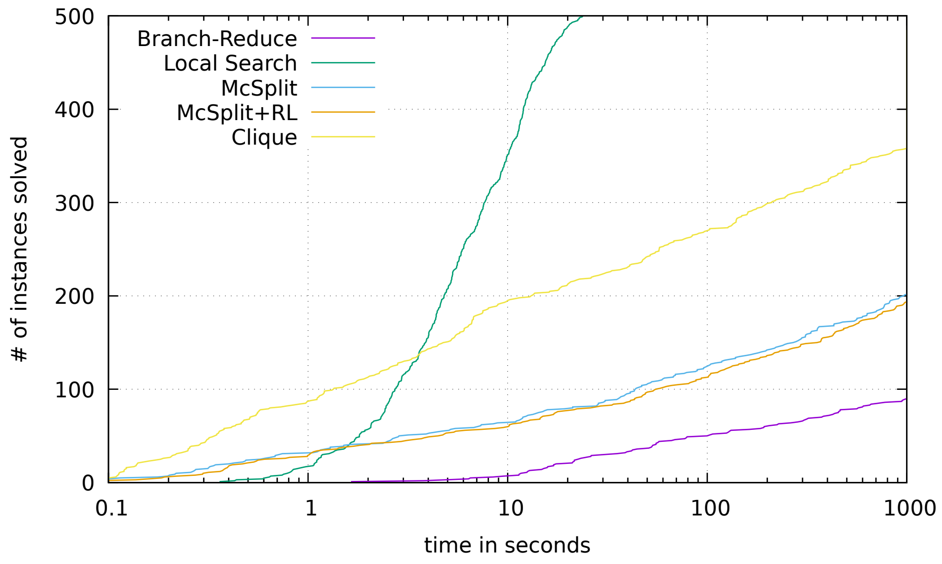
<	Branch-Reduce	Local Search	McSplit	McSplit+RL	MoMC
Branch-Reduce	x	53	24	24	59
Local Search	2	x	14	14	29
McSplit	208	226	x	52	161
McSplit+RL	220	237	74	x	171
MoMC	116	122	50	55	x

Table 5.7: How many instances of the `mcsved` set was an algorithm able to solve with a larger solution than the other algorithms. Read: `ROW` achieved a smaller result on `X` instances compared to `COLUMN`.

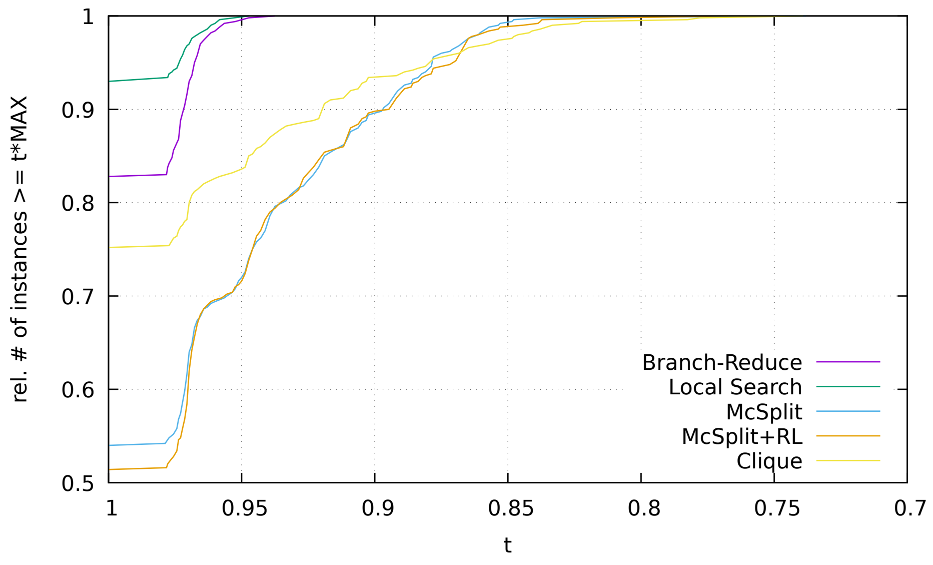
MoMC struggles to find good solutions on a few instances, at the worst-case finding solutions of only 59% of the maximum, whereas the Branch-And-Reduce solver at the worst-case finds solutions of 88% of the maximum and for McSplit, McSplit+RL and the Local Search solver the solution size is at least 90% of the maximum. Overall, McSplit and McSplit+RL perform best, finding the maximum on roughly 450 instances. The Local Search finds the maximum on only about 320 instances, for MoMC and the Branch-And-Reduce solver it is 205 and 160 instances respectively. In the direct comparison in Table 5.6, we can see that the Branch-And-Reduce solver finds solutions smaller than those of MoMC on 74 instances, whereas vice-versa it is 245 instances. Thus clearly indicating, that the solutions by MoMC are overall inferior. Additionally, this table shows that both McSplit and McSplit+RL are vastly superior on the `mcsplain` set in terms of result quality as well, having the fewest instances, for which another solver performed better. Interestingly, there are 32 instances where McSplit performs better than McSplit+RL and 31 instances vice-versa, indicating, that both algorithms are better suited for different instances.

Finally, we look at the `mcsved` instances. For this instance group, McCreesh et al. [MPT17] showed, that McSplit is inferior to a state-of-the-art clique solver. Looking at Figure 5.34a, we can confirm this, MoMC clearly outperforms both McSplit and McSplit+RL. While McSplit and McSplit+RL are able to solve close to 200 instances within the timeout, MoMC is able to solve over 350 instance. The Branch-And-Reduce solver is only able to solve slightly fewer than 100 instances, and the running time overall is fairly slow as well. However, the Local Search solver is able to solve all instances within less than 30 seconds, thus outperforming all other approaches significantly. In terms of result quality, Figure 5.34b shows that again the Local Search and Branch-And-Reduce solver perform better than MoMC. Here, the Local Search is able to solve within at least 95% of the maximum and it finds the maximum for about 470 instances. The Branch-And-Reduce solver finds the maximum on 414 instances, whereas for MoMC it is only 376. The clique solver outperforms both McSplit and McSplit+RL in terms of result quality as well, achieving the maximum solution size on many more instances, as McSplit and McSplit+RL find the maximum on only 270 and 257 instances, respectively. Though again, there are a few single instances, for which MoMC struggles to find good solutions, and thus the worst-case solution quality drops to 74% of the maximum. Looking at Table 5.7, we can again see that the Local Search solver performs best and both McSplit and McSplit+RL perform worst.

In summary, over all instance groups McSplit and McSplit+RL exhibit the best running time behavior, only for vertex- and edge-labeled graphs do the clique and Local Search solver outperform both algorithms significantly. Given our changes made to the algorithms, both McSplit and McSplit+RL perform exceptionally well on `COX2` instances, where both algorithms are able to solve all instances to optimality within at most six seconds, whereas all other solvers struggle immensely on that instance group. The Branch-And-Reduce solver is slowest overall, leading to the most timeouts by far, however, in terms of result quality, this solver mostly performs surprisingly well, outperforming the clique solver on most instance groups. In terms of result quality, the clique solver achieves the worst performance overall, as for all instance groups there are a few instances, where the clique solver is unable to find good results within the set



(a) Instances solved over time



(b) Result quality

Figure 5.34: Comparison of running time and solution quality on `mcsved` instances for the best algorithms.

timeout. The Local Search solver performs overall quite well, especially on the hard vertex- and edge-labeled instances of `mcsved` and the larger `proteins` instances, the Local Search vastly outperforms all other solvers in terms of result quality, for `mcsved` instances, the Local Search significantly outperforms the other solvers in terms of running time as well.

6 Discussion

6.1 Conclusion

In this work we discussed the MCS problem, where we want to find the largest common subgraph of two given input graphs. We introduced novel methods of reducing the problem size, using reductions on the input graphs. The reductions introduced appear to be too specific and only apply on easy instance, thus they are of little use in practice. We additionally proposed a new method of computing upper bounds on the product graph or its complement. In our experimentation we applied this upper bound to the product graph complement and found a few instances, where the improved upper bound was equal to the maximum solution size. However, the bound only improved significantly for easy instances. As a means to reduce the search size, we evaluated the exploitation of automorphisms of the input graphs and the product graph complement. We have shown, that an exact exploitation is not useful and may in the worst-case even hinder the performance in terms of running time as well as result quality. However, we have shown that our heuristic exploitation performs very well in practice, in most cases outperforming the baseline algorithms both in terms of running time and result quality. Finally, we proposed improvements to existing solvers, in order to improve their performance. For McSplit+RL we introduced two different means of computing initial scores for the reinforcement learning and we were able to show the benefit of these initial scores. For one instance group, the speedup is quite significant, for the other instance groups the speedup was less pronounced, however, result quality increased in almost all cases. For the independent set solvers of KaMIS we proposed an alternative reduction style, which may improve the performance of the reductions significantly. Additionally, for the Local Search solver we proposed an iteration limit, which may terminate the search earlier than the default would, thus improving the running time.
In our final evaluation, we compared the KaMIS solvers, with McSplit, McSplit+RL and the clique solver MoMC. Our experiments confirmed the results of McCreesh et al. [MPT17], who showed that McSplit performs better than a clique solver on most graph classes. However, on vertex- and edge-labeled and directed graphs our experiments confirmed, that McSplit is inferior to a clique solver. The heuristic Local Search solver proves to be superior on the vertex- and edge-labeled `mcsved` instances, both in terms of running time and result quality. For the `proteins` instances, the Local Search is superior in terms of result quality as well. The Branch-And-Reduce solver of KaMIS is mostly inferior on all instance groups in terms of running time. However, when we looked at the result qualities, we observed stark differences between the various solvers. We mostly see, that the clique solver MoMC struggles immensely on some instances, achieving only very small results. Thus the overall result quality of MoMC is worse than that of any other solver. This result is surprising, seeing as MoMC often achieves worse quality than the Branch-And-Reduce solver, even if the latter times out on far more instances. Additionally, we have not seen such a quality comparison before in the literature, where the focus mainly lies on running time. It is therefore of further interest to have the quality comparisons in this work.

6.2 Future Work

We proposed reduction rules for reducing the input size of the graphs, however, it seems that these rules mainly apply on simple instances, and thus overall have little effect on the running time of the solvers. For future work, it may be interesting to further investigate these rules and try to find further or more general rules.

Regarding McSplit+RL, we proposed an initialization phase for computing initial scores for all vertices. Our method accumulates scores for a vertex based on all vertices of the other input graph. An alternative method may be to compute scores specifically for pairs of vertices – one vertex from each input graph – and to prioritize pairs with largest scores during the branching. Our experiments indicate that a heuristic independent set solver can outperform exact solvers with regards to result quality on difficult instances. We therefore propose further investigating the heuristic solver in the context of even harder and larger instances. Additionally, we may investigate the use of the heuristic solver as a means of computing a lower bound for exact solvers.

Bibliography

[AKSRL07] Faisal N. Abu-Khzam, Nagiza F. Samatova, Mohamad A. Rizk, and Michael A. Langston. The Maximum Common Subgraph Problem: Faster Solutions via Vertex Cover. In *2007 IEEE/ACS International Conference on Computer Systems and Applications*. IEEE, 5 2007.

[BFG$^+$02] Horst Bunke, Pasquale Foggia, Corrado Guidobaldi, Carlo Sansone, and Mario Vento. A Comparison of Algorithms for Maximum Common Subgraph on Randomly Connected Graphs. In *Lecture Notes in Computer Science*, pages 123–132. Springer Berlin Heidelberg, 2002.

[BK73] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 09 1973.

[CFV07] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging Complexity of Maximum Common Subgraph Detection Algorithms: A Performance Analysis of Three Algorithms on a Wide Database of Graphs. 11(1):99–143, 2007.

[CVM77] Michael M Cone, Rengachari Venkataraghavan, and Fred W McLafferty. Computer-aided interpretation of mass spectra. 20. molecular structure comparison program for the identification of maximal common substructures. *Journal of the American Chemical Society*, 99(23):7668–7671, 11 1977.

[DCH97] S. Djoko, D.J. Cook, and L.B. Holder. An empirical study of domain knowledge and its benefits to substructure discovery. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):575–586, 1997.

[ER11] Hans-Christian Ehrlich and Matthias Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: a review. *WIREs Computational Molecular Science*, 1(1):68–79, 1 2011.

[FGK09] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM*, 56(5):1–32, 8 2009.

[GARW93] Helen M. Grindley, Peter J. Artymiuk, David W. Rice, and Peter Willett. Identification of Tertiary Structure Resemblance in Proteins Using a Maximal Common Subgraph Isomorphism Algorithm. *Journal of Molecular Biology*, 229(3):707–721, 2 1993.

[GFM$^+$14] Steven Gay, François Fages, Thierry Martinez, Sylvain Soliman, and Christine Solnon. On the subgraph epimorphism problem. *Discrete Applied Mathematics*, 162:214–228, 1 2014.

[GJ78] M. R. Garey and D. S. Johnson. Strong NP-Completeness Results. *Journal of the ACM*, 25(3):499–508, 7 1978.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[HMR17] Ruth Hoffmann, Ciaran Mccreesh, and Craig Reilly. Between Subgraph Isomorphism and Maximum Common Subgraph. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 3907–3914. AAAI Press, 02 2017.

[Jac12] Paul Jaccard. The Distribution of the flora in the Alpine zone. *New Phytologist*, 11(2):37–50, 2 1912.

[Koc01] Ina Koch. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science*, 250(1-2):1–30, 01 2001.

[Lev73] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341–352, 12 1973.

[LJM17] Chu-Min Li, Hua Jiang, and Felip Manyà. On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem. *Computers & Operations Research*, 84:1–15, 8 2017.

[LLJH20] Yanli Liu, Chu-Min Li, Hua Jiang, and Kun He. A Learning Based Branch and Bound for Maximum Common Subgraph Related Problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(03):2392–2399, 4 2020.

[LSS$^+$19] Sebastian Lamm, Christian Schulz, Darren Strash, Robert Williger, and Huashuo Zhang. Exactly Solving the Maximum Weight Independent Set Problem on Large Real-World Graphs. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 144–158. Society for Industrial and Applied Mathematics, 1 2019.

[MFK21] Christopher Morris, Matthias Fey, and Nils M. Kriege. The Power of the Weisfeiler-Leman Algorithm for Machine Learning with Graphs. *CoRR*, abs/2105.05911, 2021.

[MKB$^+$20] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. TUDataset: A collection of benchmark datasets for learning with graphs. *CoRR*, abs/2007.08663, 2020.

[MNPS] Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and Constraint Models for Maximum Common (Connected) Subgraph Problems. In Michel Rueher, editor, *Lecture Notes in Computer Science*, Lecture Notes in Computer Science, pages 350–368. Springer International Publishing.

[MP03] Alessio Massaro and Marcello Pelillo. Matching graphs by pivoting. *Pattern Recognition Letters*, 24(8):1099–1106, may 2003.

[MP14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 1 2014.

[MPT17] Ciaran McCreesh, Patrick Prosser, and James Trimble. A Partitioning Algorithm for Maximum Common Subgraph Problems. In *Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 712–719, 2017.

[NS11] Samba Ndojh Ndiaye and Christine Solnon. CP Models for Maximum Common Subgraph Problems. In *Principles and Practice of Constraint Programming – CP 2011*, pages 637–644. Springer Berlin Heidelberg, 2011.

[PRS13] Younghee Park, Douglas S. Reeves, and Mark Stamp. Deriving common malware behavior through graph clustering. *Computers & Security*, 39:419–430, 11 2013.

[RW02] John W. Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[SAKZ$^+$05] W. Henry Suters, Faisal N. Abu-Khzam, Yun Zhang, Christopher T. Symons, Nagiza F. Samatova, and Michael A. Langston. A New Approach and Faster Exact Methods for the Maximum Common Subgraph Problem. In *Lecture Notes in Computer Science*, pages 717–727. Springer Berlin Heidelberg, 2005.

[SFSV03] M. De Santo, P. Foggia, C. Sansone, and M. Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. 24(8):1067–1079, 05 2003.

[SSvL+11] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *J. Mach. Learn. Res.*, 12:2539–2561, 2011.

[TGL+19] Matteo Togninalli, M. Elisabetta Ghisu, Felipe Llinares-López, Bastian Rieck, and Karsten M. Borgwardt. Wasserstein Weisfeiler-Lehman Graph Kernels. *CoRR*, abs/1906.01277, 2019.

[VV08] Philippe Vismara and Benoît Valery. Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms. In Hoai An Le Thi, Pascal Bouvry, and Tao Pham Dinh, editors, *Communications in Computer and Information Science*, Communications in Computer and Information Science, pages 358–368. Springer Berlin Heidelberg, 2008.

[War16] Matthijs J. Warrens. Inequalities Between similarities for numerical Data. *Journal of Classification*, 33(1):141–148, 4 2016.

[WL68] Boris Weisfeiler and Andrei Leman. The reduction of a graph to canonical form and the algebra which appears therein. *NTI, Series*, 2(9):12–16, 1968.