Engineering Local Search for the Maximum Independent Set Problem on Hypergraphs

Patricia Oehler

March 1, 2026

4279576

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisor:
Dr. Ernestine Großmann

Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Christian Schulz for giving me the opportunity to write my thesis in his research group and for his quick responses. I am very grateful to Dr. Ernestine Großmann for the time she invested to support me whenever I needed it. I learned a lot from her and was inspired by our long discussions.

Eigenständigkeitserklärung
Vorname, NACHNAME: Patricia Oehler
Matr.-Nr. 4279576
Hiermit versichere ich, dass ich die von mir am ..02.03..2026..als PDF-Datei (mit Standard-PDF-Lesesoftware lesbar und durchsuchbar) eingereichte Bachelorarbeit mit dem Titel "Engineering Local Search for the Maximum Independent Set Problem on Hypergraphs"
betreut von Prof. Dr. Christian Schulz selbstständig und ohne unzulässige fremde Hilfe verfasst habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und sämtliche Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Werken - einschließlich solcher aus digitalen oder KI-basierten Quellen - übernommen wurden, als solche kenntlich gemacht. Ich bestätige, dass die etwaige Nutzung von KI-Tools vorab mit der Betreuerin/dem Betreuer der Abschlussarbeit abgesprochen wurde und dass die besprochenen Regelungen eingehalten wurden. Ich übernehme die volle Verantwortung für die wissenschaftliche Qualität und den Inhalt der vorliegenden Arbeit, die gewählte Methodik und den Erstellungsprozess, sowie die zitierte Literatur. Ich bin damit einverstanden, dass meine Arbeit im Rahmen universitärer Verfahren auf Plagiate sowie auf den möglichen Missbrauch automatisierter Text- und Codegenerierung überprüft wird.

..Heidelberg, 02.03.2026.. ....[illegible]............
Ort, Datum Unterschrift
The generative artificial intelligence tool ChatGPT (OpenAI) was used to assist the generation of plots, the formatting of tables, and the verification of short code snippets. All outputs were critically reviewed and verified to ensure correctness.
Patricia Oehler

Abstract

Finding maximum independent sets is an important and widely investigated NP-hard optimization problem. An independent set is a subset of vertices of a graph that are pairwise not adjacent to each other. The largest independent set on a graph is the maximum independent set (MIS). There are many approaches to solving the MIS problem. Heuristic strategies have proven to be very effective in quickly finding high-quality solutions to the problem for large instances that are hard to solve exactly. Though there is much focus on finding large independent sets on graphs, little work has been conducted for the MIS problem on hypergraphs, for which edges may include an arbitrary number of vertices. In this work, we implement an iterative local search algorithm based on 2-improvements to approximate MIS. In more detail, we consider the effect of reductions on the algorithms' performance. A reduction in this context is an operation that transforms a graph to a smaller instance according to specific rules. Previous work on the topic shows that reducing a graph instance before performing an algorithm can significantly improve the performance of exact and heuristic approaches. We apply reductions in between steps of the local search to improve the overall efficiency. This online manner of execution is meant to balance the positive effect achieved by reductions and the time overhead they produce. For the online reductions, we utilize a number of previously presented reductions for hypergraphs. We add an inexact reduction and a vertex folding reduction initially proposed for graphs and apply them to the structure of hypergraphs. Compared to local search algorithms on graphs, we mostly find larger solutions for our hypergraph-specific local search in less time. Applying online reductions yields even better results. It provides sizes larger by up to 0.2% of the best solution for one instance, averaging it over multiple runs of the algorithm. The average time to reach the best solution an algorithm finds within a run is shorter by about a hundred seconds. With our online reductions, we reduce the average hypergraph to 55% of its original size within 3.59 seconds on average. We further extend a hypergraph preprocessing in this thesis, which, in combination with the online reductions, reduces the average hypergraph to 24% of its original size within 7.89 seconds.

Contents

CHAPTER 1

Introduction

A maximum independent set (MIS) for a graph $G = (V, E)$ is a maximum cardinality set $I \subseteq V$ of vertices that are pairwise not adjacent to each other. Finding maximum independent sets is an important and widely investigated NP-hard optimization problem.
This thesis implements an iterative local search to approximate MIS on hypergraphs. In hypergraphs, edges can include any number of vertices instead of just two. We discuss the possible improvement of the local search through reduction techniques.
We first depict a local search based on $(1, 2)$-swaps, following the algorithm proposed by Andrade et al. [3]. A $(1, 2)$-swap removes a node from the solution I, allowing two others to be inserted, thus increasing the size of the solution one at a time.
We furthermore focus on the application of reductions during the iterative local search process. Here, a reduction transforms a graph into a smaller instance, by e.g. removing a vertex, according to specific rules in a way that does not change the solution. The approach is inspired by the success of online-fashion reductions in the local search by Dahlum et al. [9]. We adopt several of the reductions used by Dahlum et al., including some initially described by Akiba and Iwata [1], as well as additional ones proposed by Großmann et al. [19], focusing on relatively simple techniques.

1.1 Motivation

The application of maximum indepent sets stretches over many fields. For instance, it is used in social networks [10], in biology, e.g. to find similarities in DNA [24], and for identifying critical nodes in wireless networks [14].
Hypergraphs are often used to represent high-order correlations that are otherwise not portrayable in graphs [15], for example when modeling biological networks [13]. In such cases, algorithms specifically designed for hypergraphs are advantageous.
We aim to improve performance by avoiding conversion of hypergraphs to graphs, which is especially hard for highly connected hypergraphs, and making use of the structure. Specif-

ically, for the solving of the maximum independent set problem, the hypergraph structure is favorable. For example, large cliques, where each vertex is connected to all others in the clique, hold up the process of finding maximum independent sets, as only one vertex within the clique can be in the solution. Combining them into a single hyperedge facilitates many operations and gives better heuristics.
Being NP-hard, finding and approximating maximum independent sets is especially difficult on large graphs with complex structures. Heuristic approaches such as iterative local search have proven to be powerful in quickly finding large independent sets on such graphs [3]. Reducing instances that are hard to solve to smaller, more easily solved ones has furthermore yielded highly positive results, improving the performance of algorithms operating on reduced graphs [19].
The complexity and effectiveness of the selected reductions varies, so preprocessing the graph by reducing it before the execution of an algorithm can take much time. We want to balance the time overhead of reductions and the gain achieved by their application by instead reducing the graph step-by-step in between the operations of the local search.

1.2 Our Contribution

In this thesis, we implement an iterative local search for the MIS problem specific for hypergraphs. We further implement reductions in order to improve the performance of the local search. For this, we apply online reductions that are executed in between operations of the local search. These include exact and inexact reductions. We use previously defined reductions, some already designed for hypergraphs. We derived the inexact reduction and the vertex folding reduction from similar reductions on graphs. We show that relatively simple reductions are a good choice for the implementation. The specific manner of the incorporation of the reductions into the local search influences the performance greatly, and we give options for places within the algorithm to execute them.
We conduct extensive experiments to fine-tune the different options and other parameters with which we adjust the reductions. Further experiments compare our algorithm to two local search algorithms for graphs and an ILP for hypergraphs. In this project, we extend a preprocessing by consecutively executing one with all exact reductions we implement. We test the different algorithms in combination with this preprocessing. In the experiments, we use a constant time limit and focus on maximizing the solution quality. To find the highest-quality solutions, we must balance the time a reduction consumes and the speed up it results in.
To approximate MIS on hypergraphs, local search outperforms exact solvers on large sparse graphs that are often not exactly solvable in a given time. We still find exact solutions for many instances where the exact approaches successfully solved a hypergraph. Moreover, we find that online reductions can significantly speed up the process, leading to solutions better by up to 0.2% for a single instance in roughly 100 seconds less time for the average instance. Compared to the preprocessing, we spend half as much

time on online reductions alone and reduce the hypergraph by 15% of the original hypergraph size less than the preprocessing. Our resulting algorithms compete with graph local searches time- and quality-wise, yielding a higher solution quality and faster times when no preprocessing is conducted.

1.3 Structure

The remainder of this thesis is organized as follows. In the 'Fundamentals' Chapter 2, we explain important notions in the context of the problem of this work. In Chapter 3, we give an overview over related work on the topic. We continue with the 'Algorithm Description' Chapter 4, where we explain the different parts of the procedure in more detail, covering the basic execution of the local search, the incorporation of online reductions and our addition to the preprocessing. In the 'Experimental Evaluation' Chapter 5, we test different configurations for our local search and evaluate the performance of the best configuration in combination with the preprocessing. We then compare our algorithm to two different local search algorithms on graphs and an ILP for hypergraphs. Lastly, the Chapter 6 'Discussion' lists the most important findings and then gives an outlook for future work on the topic.

CHAPTER 2

Fundamentals

2.1 General Definitions

Given an undirected graph $G = (V, E)$, $|V| = n$ is the number of vertices in G and $|E| = m$ is the number of edges in G. An edge e is incident to a vertex v, when $v \in e$ and we define the degree of a vertex $deg(v)$ as the number of incident edges of v. $N(v) = \{u : (v, u) \in E\}$ is called the open neighborhood of a vertex v. The size of the largest neighborhood is defined as $\Delta = max_{v \in V}(|N(v)|)$. Contrary to graphs, where for a graph $G = (V^G, E^G)$ an edge $e \in E^G$ always connects exactly two vertices, a hyperedge $e \in E^H$ of a hypergraph $H = (V^H, E^H)$ can contain any number of vertices. For a hypergraph $H = (V, E)$ an edge $e \in E$ is a subset of vertices $e \subseteq V$ that connects all contained vertices $v \in e$. We define the size of a hypergraph $|H|$ as the sum of all edge sizes. We denote the number of vertices it contains with $|e|$. For a hypergraph H, the cardinality of the neighborhood $|N(v)|$ and the degree $deg(v)$ can differ.
The rank of a hypergraph denotes the maximum edge size $max_{e \in E}|e|$ and the hypergraph is called k-$uniform$, when each edge has size k. The graph is sparse, when $m = O(n)$.
A clique is a subset of vertices $C \subseteq V$ where each vertex $v \in C$ is connected to all other vertices $w \in C$ with $w \neq v$. A hypergraph H can be converted to a $corresponding\ graph\ G$ through clique expansion, by forming a clique out of all contained vertices $v \in e$ for each hyperedge $e \in E$.
In this thesis, we consider the strong unweighted independent sets on H, which is a subset I of V, where for any two vertices $v, w \in V$, there is no edge $e \in E$ that connects v and w. The size of the MIS on H is denoted with $\alpha(H)$.
In contrast, a weak independent set is a set of vertices $IW \subseteq V$, so that for each edge $e \in E : e \not\subseteq IW$. Weights are a value assigned to each edge (edge weights) or vertex (node weights). The maximum weighted independent set is a set $MWIS \subseteq V$ that maximizes the sum of all node weights in an independent set.

The goal of the maximum independent set (MIS) problem is finding an independent set $I \subseteq V$ with the largest possible cardinality.
The solution for an algorithm to find MIS is an independent set I with size $|I|$. The tightness $T(v)$ of a vertex v denotes the number of neighbors of the vertex that are in the solution, called "k-tight" for $T(v) = k$. A free vertex is a vertex with tightness 0.
The reductions used in this thesis assure that the maximum independent set does not change when reducing a graph, i.e. the MIS $I' \in R$ on the reduced graph R is a subset $I' \subset I$ of the MIS $I \in H$ on the original graph.
When removing a vertex v, we reduce a Hypergraph H to a hypergraph $H^R = (V^R, E)$, where $V^R = V \setminus \{v\}$. Analogously, when removing an edge e we produce a hypergraph $H^R = (V, E^R)$, where $E^R = E \setminus \{e\}$. Folding three vertices u, v and w, with u and w being the only neighbors of v, results in the reduced hypergraph $H^F = (V^F, E^F)$. Here $V^R = V \setminus \{u, w\}$ and for all edges $e \in E$ that contain u or w, the resulting edge $e' \in E^F$ instead contains v.
A $subgraph$ is a graph $H' = (V', E')$ of $H = (V, E)$, when $V' \subseteq V$ and $E' \subseteq E$.

CHAPTER 3

Related Work

Much work has been conducted in order to find large independent sets in a fast manner. The following lists important works in the topic. We first describe exact and inexact methods on graphs and discuss hypergraph applications after.

3.1 Exact Methods

As finding an MIS is NP-hard, exact calculation through exhaustive search takes exponential time. Many approaches have been tried on graphs in order to minimize the exponential running time. One common method when solving optimization problems to reduce the search space is the branch-and-bound method [28], which has been widely utilized for the finding of MIS.
Furthermore, reductions have repeatedly proven to speed up the finding of MIS.
As early as 1977, Tarjan et al. [30] successfully implemented an $O(2^{\frac{n}{3}})$ time algorithm. Related methods, as used by Bourgeois et al. [7] who found MIS in $O(1.2114^n)$, repeatedly set new records.
Combining reductions and branch-and-bound algorithms further facilitates the computation. Much larger instances than before can be solved exactly when removing elements and easily solved subgraphs before branching.
Akiba and Iwata [1] provide a successful branch-and-reduce algorithm with an upper bound of $O(1.2210^n)$, proving that more complex reductions are also effective for quickly finding solutions on large instances especially.
The current best polynomial-space algorithm by Xiao and Nagamochi [33] with time $O(1.2110^n)$ surpasses even the $O(1.2109^n)$ exponential-space algorithm Robson [29] developed when considering a time-space trade-off.
Moreover, the problem can be solved quickly for many graphs with an integer linear program as given by Großmann et al. [19], using ILP-solvers such as Gurobi [21] to calculate the optimal solution.

3.2 Heuristic Methods

Especially large graph instances remain an obstacle for exact approaches. Even small instances can be extremely difficult to solve [8].
Instead, heuristic approaches such as local search approximate high-quality solutions of the MIS problem. In practice, they have proven to be much faster in finding large independent sets for large sparse instances in particular. They even find exact solutions as well for many smaller graphs [9].
Andrade et al. [3] introduced an iterated local search (which we call ARW in the following) with particularly good results. They based their approach on previous work for the related maximum clique problem [20, 22]. The combination of $plateau\ search$ and improvements of an initial solution achieved by swaps and vertex insertions and deletions inspired their algorithm. Using (j, k)-swaps, or k-$improvements$ for $k > j$, where removing j vertices from the solution allows k vertices to be inserted, they improve the solution one vertex at a time. They iterate over a perturbation of a current solution and a scan of the neighborhood for such $(1, 2)$-swaps in a local search. This procedure illustrates a suitable basis for a similar algorithm on hypergraphs.
Lamm et al. [26] provide an evolutionary algorithm, adding heuristic reductions to exact reduction techniques. They show that removing vertices that are very likely in the solution recursively, thereby increasing the number of possible exact reductions, leads to much smaller reduced graphs. Thus, they significantly improve performance for large sprase graphs while still finding exact solutions for simpler instances.
More research has been dedicated in recent years to applying similar techniques to the maximum weighted independent set problem [17]. For example, Großmann et el. [18] introduce a local search for the maximum weighted independent set using a novel current heuristic where they apply reductions in their approach.
Dahlum et al. [9] propose a local search algorithm that is strongly based on ARW [3]. They use relatively simple reduction rules in preprocessing, which is otherwise notably expensive. Instead, the local search is accelerated by performing an online reduction of the graph. They apply exact and inexact reductions, cutting high-degree vertices, as they are unlikely to be in the solution and removing isolated vertices and their neighbors from the solution whenever inserting a vertex into it.

3.3 Application to Hypergraphs

When considering hypergraphs, practical implementation to find MIS has little to no presence in research so far. A great deal of research has previously focused on analysing the complexity of algorithms to find MIS and the quality of their solutions.
The theory often regards weak independent sets and frequently observes hypergraphs of bounded edge size [6] or degree [31] or of a specific structure, such as k-uniform hypergraphs [23]. Balliu et al. [4] present complexities dependent on maximum degree and edge

size of a hypergraph.
There is little mention of local search on hypergraphs to find MIS. Losievskaja [27] is one of the few who address the strategy, though also just in theory. Losievskaja, besides greedy and partitioning approaches, also describes a method akin to k-improvements in ARW [3] for the local search. When reaching a local optimum, this local search is proven to guarantee $\frac{\Delta}{2+\epsilon}, \lim_{t\to\infty} \epsilon(t) = 0$ on bounded degree hypergraphs, specifically $(\frac{\Delta+1}{2})$ for 2-$improvements$.

CHAPTER 4

Algorithm Description

In this chapter, we describe our algorithm to compute a high-quality solution to the MIS problem. We start with a simple greedy algorithm to compute an initial solution I for the local search. We then continue with the description of the general iterative local search (which we call LSHyMIS in the following) based on the ARW [3] algorithm. For some vertices, the local search performs especially bad, others are certain to be in an MIS or in none. To avoid unnecessary checks for those vertices, we use reduction rules that remove specific vertices from the solution. We list the reductions we used and introduce their incorporation into the iterative local search.

4.1 Greedy Algorithm

The greedy algorithm uses the fact that vertices with many neighbors are less likely to be in a large independent set, as the probability of a neighbor being in the solution increases with every neighbor. For the input hypergraph $H = (V, E)$, we sort an array of all vertices by $|N(v)|$ from smallest to largest in this algorithm. We iterate over the sorted vertex array and insert the free vertex with the next smallest count of neighbors into the solution until no free vertices are left.

Remark 1. If the value we sort by is the same for multiple vertices, we pick the vertex with the lowest ID first when sorting. This insures a clear order within the sorted structure.

4.2 Iterative Local Search

Starting with the independent set the greedy algorithm provided, we aim to improve this solution by finding $(1, 2)$-swaps in a neighborhood. In an $(1, 2)$-swap, the removal of a vertex from the solution allows two others to be inserted into the solution. We call this a

2-*improvement*. We apply the algorithm by Andrade et al. (ARW in the following) [3] to the structure of hypergraphs and describe the resulting procedure. Algorithm 1 gives an overview of the most important steps.
The iterative local search follows a general structure presented in Algorithm 1. We iterate over a sequence of *perturbation*, finding a local optimum in a *local search step* and lastly deciding whether to *keep or discard* the solution found in this step.

During the local search, the solution I is an independent set at all times. We maintain the solution as an array that specifies for each vertex whether it is part of the solution or not. We also store a block containing all solution vertices, one for all free vertices and one for all non-solution vertices, as well as the number of solution, non-solution and free nodes explicitly. Furthermore, we store an array, partitioning all nodes into these three blocks. With these structures, determining whether a node is free or in the solution or picking any node within one of the blocks can be done within $O(1)$. This is very useful for many procedures in the following. Furthermore, we sort the neighborhood by degree as described in Remark 1.

Step 1: To perturb the solution, we pick one non-solution vertex v or with a probability of $\frac{1}{(2S)}$, we pick $i = \log_2(2S)$ vertices, and force v into the solution. Forcing a vertex into the solution is done by first removing all solution neighbors from the solution, so the vertex becomes free and inserting it into the solution afterward. When picking multiple

Algorithm 1 LSHyMIS

```
1:  procedure ITERATIVELOCALSEARCH($H = (V, E^H),\ I_{Greedy}$),
2:      $S,\ bestS \leftarrow I_{Greedy},\ tightness \leftarrow T(v)$ for each vertex,
3:      $candidates \leftarrow$ all solution vertices
4:      sortNeighborhood($H$)
5:      while runningTime < timeLimit do
6:          if $|S| > |bestS|$ then
7:              $bestS = S$
8:          end if
9:
10:         $S' \leftarrow$ perturbationStep()                                   ▷ Step 1
11:         $S' \leftarrow$ localSearchStep()                                   ▷ Step 2
12:         if $|S| > |S'|$ then
13:             continue for $S$ within $|S|$ iterations of going to a worse solution
14:         else
15:             continue for $S = S'$
16:         end if
17:     end while
18:     return $S$
```

vertices to force into the solution, we choose each vertex within a distance of exactly two to all previously chosen vertices to be forced into the solution. We force only the vertices chosen so far if there is no other such vertex left in the hypergraph. We iteratively insert all vertices that have become free during the forcing process into the solution.
For diversification in the local search step, we scan the (first) vertex that was forced into the solution in the preceding perturbation only after all other candidates were scanned already.

Step 2.1: An individual local search step — as illustrated in Algorithm 2 — finds a local optimum within the neighborhood of the forced vertex by scanning all vertices within a list of candidates for $(1, 2)$-swaps in random order. We store a partition of vertices, indicating for each vertex whether it is a candidate, as well as a candidate and a non-candidate block, so that we can pick a random candidate or determine for a vertex whether it is a candidate in $O(1)$.
To avoid scanning all vertices in every iteration, where we would repetitively check for $(1, 2)$-swap on unaltered neighborhoods, we instead only scan a list of candidates. We start with all solution vertices being candidates and remove them from the list of candidates if they were scanned for 2-improvements, and after they are removed from the solution. We insert a vertex into the list of candidates when it is inserted into the solution or when the tightness of a neighbor is reduced to 1.
Step 2.2: Scanning a solution vertex v for a $(1, 2)$-swap (Figure 4.1) is done by first removing it from the solution. All 1-tight neighbors, for which v was the only solution neighbor before, become free after removing v from the solution. It is only possible to exchange v with two of its neighbors in the solution if at least two vertices besides v have become are free when removing it. To check whether we find a $(1, 2)$-swap, we insert a free neighbor n_1 of v into the solution. If there are other free hypernodes left after that, a $(1, 2)$-swap is possible, and we iteratively insert all remaining free vertices into the solution. If not, we remove n_1 again and insert another free neighbor until all were checked. In case we find no $(1, 2)$-swap for all neighbors, no 2-improvement is possible for the currently scanned vertex, and it is inserted back into the solution. We remove it from the candidates list as it was unsuccessfully scanned and continue for another random candidate. We stay in the neighborhood for as long as possible, if a $(1, 2)$-swap happened

Algorithm 2 Scan Candidates for $(1, 2)$-swaps

```
1: procedure LOCALSEARCHSTEP
2:     for $currentNode \in candidates$ do
3:         check whether a $(1, 2)$-swap is possible for $currentNode$        ▷ Step 2.1
4:         if swap happened then
5:             choose next candidate within $N(currentNode)$ if possible      ▷ Step 2.2
6:         end if
7:     end for
8:     return $S$
```

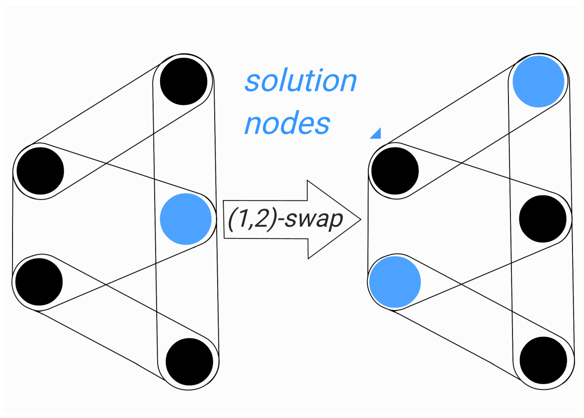


Figure 4.1: $(1, 2)$-swap

and scan the next candidate in the neighborhood until no candidate neighbor is left.

We scan each vertex $O(1)$ times to find 2-improvement during the local search step. A vertex is inserted and/or removed as a candidate itself or as a 1-tight neighbor during the scanning of its solution neighbor, which takes $O(|N(v)|)$ time. Because of this, finding a 2-improvement or showing that none is possible can be done in $O(n \cdot \max_v(|N(v)|))$.
We always continue for a solution if an iteration increased the size to the independent set. For a decreased solution size after an iteration compared to the size in the foregoing iteration, we continue for this worse solution once and then reverse any iterations leading to a smaller solution within the next $|S|$ iterations. For whenever the solution becomes smaller, we maintain a $best\ solution$, which is the largest solution that has been found throughout the whole iterative local search so far. We update it whenever we find a better solution.
The local search stops when a given time limit or a given number of iterations is reached and $best\ solution$ is returned.

4.3 Reductions

Certain vertices slow the local search down significantly. Scanning for $(1, 2)$-swaps takes much longer when a vertex with a large neighborhood is involved, as the running time of a scan for a $(1, 2)$-swap is dependent on $|N(v)|$ for a vertex v. Other vertices are guaranteed to be a part of an MIS. Testing for such a vertex whether it can be removed from the solution or forcing a vertex that is never part of any MIS into the solution is unnecessary. Instead, we remove the vertex from the graph and either include or exclude it from the solution permanently within the local search.
Removing a vertex v from the hypergraph $H = (V, E)$ and the solution I results in a reduced graph $H^R = (V^R, E)$ with $V' = V \setminus \{v\}$. If v was part of the solution before, then $I^R = I \setminus \{v\}$ with $|I^R| = |I| - 1$. Otherwise, the solution remains unchanged, yielding $I^R = I$.

For all exact reductions, we guarantee that they do not change MIS on the original hypergraph H. Furthermore, I is an independent set on H at any time and we remove a vertex v in a way that ensures that I^R is an independent set on H as well as H^R. We either exclude v from the solution, $include$ it or $fold$ three vertices into one.
1. When $excluding$ a vertex from the solution, if it was part of the solution I before, we first remove it from the solution. This ensures that when removing it from H, the tightness of all neighbors of v remains unchanged and $I^R \subseteq V^R = V \setminus \{v\}$ is an independent set.
2. We $include$ a vertex v into the solution, by first excluding all of its neighbors and then including v. By excluding all neighbors, we assure that there cannot be a vertex $w \in V \setminus N(v)$ in the H or $H^R = (V \setminus N(v), E)$ that is adjacent to v in the original hypergraph.
3. To fold the three vertices v, u, w we exclude w and u and replace them with v in all edges incident to u and w and in the neighborhoods of all neighbors $n \in N(w) \cup N(u)$, resulting in the reduced hypergraph $H^R = (V \setminus \{u, w\}, E')$. As v has only u and w, which were excluded, as neighbors in the unreduced hypergraph, this does not change the independent set.

Removing an edge e from the hypergraph $H = (V, E)$ yields the reduced hypergraph $H^R = (V, E^R)$ with $E^R = E \setminus \{e\}$. The solution I^R remains the same, as we only remove an edge, if its absence does not change the adjacency of the hypergraph, therefore not impacting any independent set. Removing a hyperedge leads to a less complex structure and decreases the degree of contained vertices, in turn assisting the success of vertex reductions.

We add for each reduction the relation of the size of the MIS $\alpha(H)$ on the hypergraph H before the reduction and the MIS size $\alpha(H')$ on the reduced hypergraph H'. We further describe the relation of the independent set I on the original hypergraph and I' on the reduced hypergraph.

The reductions were chosen balancing their effectiveness and their complexity. We need reductions that do not hinder the local search too much when executing them in between steps.
In the following section, we explain the reductions we use in our algorithm. We started by picking exact reductions that can be checked in $O(1)$: removing certain vertices with $|N(v)| < 2$ or $deg(v) < 2$ and edges with $|e| < 2$ [19]. We continued by adding more complex exact yet still relatively simple reductions: vertex folding [1], edge and vertex domination [19]. Lastly, we added an inexact reduction as proposed by Dahlum et. al. [9]. We did not choose reductions that use complex structures such as the twin and sunflower reduction [19] or isolated cliques [9] with a size larger than two (degree-zero and degree-one vertices can be seen as isolated cliques). Due to the increased connectivity on hypergraphs compared to graphs, finding such structures with in a hypergraph takes much longer with increasing hyperedge sizes.

We now give an overview over the reductions and their part in the algorithm.

4.3.1 Exact Reductions

Simple Reductions

We maintain the degree and number of neighbors for each vertex, and the size for each edge, so we can check if any of the listed simple reductions is possible for an edge or a vertex with a simple $O(1)$ operation. More time is needed only when the reduction is possible and the element is removed.

Size-Zero-Edge [19]
When removing vertices from the graph, edges can become empty. An edge e can be removed from the hypergraph, if $|e| = 0$. This yields $I' = I$ and $\alpha(H') = \alpha(H)$.
Proof: An edge that connects no vertices does not affect the adjacency of the hypergraph and can be easily removed.
Complexity:
Perform a simple $O(1)$ check.

Size-One-Edge [19]
An edge $e \in E$ can be removed from the graph, if $|e| <= 1$ and the removal results in $I' = I$ with $\alpha(H') = \alpha(H)$.
Proof: When connecting no vertices, the adjacency remains undisturbed by its removal. See also [19]
Complexity:
Perform a simple $O(1)$ check.

Degree-Zero-Vertex [19]
A vertex $v \in V$ can be included into the solution, if $deg(v) = 0$. If included, $\alpha(H') = \alpha(H) - 1$ and $I = I' \cup \{v\}$.
Proof: A vertex with no incident edges has no neighbors and is therefore not adjacent to any solution vertex $\rightarrow$ it can be included into the solution. See also [19]
Complexity:
If the $deg(v) = 0$ for a vertex v, we include it into the solution. As it is not connected to any other vertex and has no incident edges, we can simply remove it in $O(1)$.

Zero-Neighbors-Vertex
A vertex $v \in V$ can be included into the solution, if $|N(v)| = 0$ (incident edges of v might have size one, so it is not removed by the degree-zero reduction). If included, $\alpha(H') = \alpha(H) - 1$ and $I = I' \cup \{v\}$.
Proof: A vertex with no neighbors is not adjacent to any solution vertex $\rightarrow$ it can be included into the solution.

Complexity:
Removing the vertex with $|N(v)| = 0$ is in $O(deg(v))$, as we need to update all incident edges which have size 1.

Degree-One-Vertex [19]
A vertex $v \in V$ can be included into the solution, when $deg(v) = 1$. If included, $\alpha(H') = \alpha(H) - 1$ and $I = I' \cup \{v\}$.
For the **proof**, see Großmann et al. [19].
Complexity:
When the $deg(v) = 1$ for a vertex v, we include it into the solution. Its neighborhood — all vertices within the edge e — is excluded and the neighborhoods and incident edges of all of them must be updated.
This can consume much time for graphs with many large edges and vertices with few incident edges, which is why we set a limit h for the size of the edge and check before the removal, whether $|e| < h$.
Our specific implementation especially elevated the time overhead, as we remove every vertex separately from the edge, which could be done more efficiently by conducting a combined operation for removing all vertices in this edge.

One-Neighbor-Vertex
A vertex $v \in V$ can be included into the solution, when it has only one neighbor (multiple edges might be connecting the two, or v might have incident edges with size one and v is not removed by the degree-one reductions). If included, $\alpha(H') = \alpha(H) - 1$ and $I = I' \cup \{v\}$.
Proof: Analogous to the proof for degree-one by Großmann et al. [19]
Complexity:
Removing the vertex with $|N(v)| = 1$ with $N(v) = w$ is in $O(deg(v))$. Additionally, we need to remove w, updating all of its neighbors and incident edges.

4.3.2 More Complex Reductions

Vertex Folding
For a vertex v with $|N(v)| = 2$ whose neighbors $u, w \in N(v)$ are not adjacent to each other, either v or u and w are in an MIS. We can $fold$ the vertices v, u and w into a single vertex, as depicted in Figure 4.2, and decide in the end, which vertices are in the solution. The folding results in $\alpha(H') = \alpha(H) - 1$ and either $I = I' \cup \{v\}$ or $I = I' \cup \{u, w\}$. We derived this reduction from the vertex folding proposed by Akiba and Iwata [1].
Proof: Given is a vertex v with two neighbors u, w that are not adjacent to each other. There are three cases:
1. Neither u nor w are in an MIS, v is in the MIS, as they are its only neighbors.
$\rightarrow v$ is in an MIS
2. One vertex of either u or w is in an MIS, we can exchange the vertex with v in the MIS,

as it would be the only solution neighbor of v and the size of the MIS does not change.
$\rightarrow v$ is in an MIS
3. Both u and w are in an MIS and instead inserting v would reduce the size of the independent set.
$\rightarrow u$ and w are in an MIS
The cardinality of the MIS on the original graph is larger than the cardinality of the MIS of the reduced graph by one. This is because when the vertex v is in the solution I' on the reduced instance, it is exchanged with both u and w in the independent set I on the unreduced hypergraph, yielding $|I| = |I'| + 1$. Otherwise, if v is not in I', w and u are not in I. Being the only two neighbors of v, v is in I in that case, again leading to $|I| = |I'| + 1$.
Complexity:
We check for a vertex v in the hypergraph $H = (V, E)$ if $|N(v)| = 2$ and $deg(v) > 1$. For $deg(v) = 1$ the two neighbors $u, w \in N(v)$ are adjacent. We then check whether u and w are connected. This takes $O(log(max(|N(u)|, |N(w)|)))$, as we are operating on a sorted neighborhood, and can perform a binary search on the neighborhood. Overall, checking whether vertices can be folded can be done in $O(log(max(|N(u)|, |N(w)|)))$ in total.
In the end, if v is in the solution, there are no solution vertices in the neighborhood $N(v)$ for the reduced hypergraph H^R, consequently there are no solution vertices in $N(w) \cup N(u)$ for the original hypergraph H. Thus, u and w are both in the solution and v is not. Otherwise, v is inserted into the solution and u and w remain outside.
When conducting multiple recursive foldings, we decide in reverse order of their execution.

Remark 2: In the following reductions, we check whether a sorted set of elements A is subset of another sorted set B by iterating over both sets at the same time, continuing to the next element of the possible subset A only when the same element appears in B. This is done in time linear to the size of the smaller set.

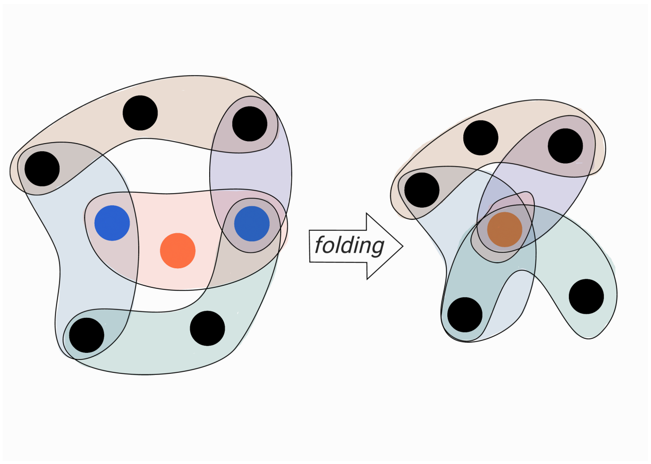


Figure 4.2: Folding Reduction

Vertex Domination [19]
A vertex v dominates by a neighboring vertex w, if $N(w) \subseteq N(v)$. v can be excluded from the solution. If excluded, $I' = I$ and $\alpha(H') = \alpha(H)$.
The **proof** is given by Großmann et al. [19].
Complexity:
We check if a vertex v dominates one of its neighbors w by checking if $N(w)$ is a subset of $N(v) \cup v$ as described in Remark 2. As the neighborhoods are sorted, this can be done in $O(min(|N(v)|, |N(w)|))$.
Especially for vertices with many neighbors, this takes long. Hence, we only check this for vertices v with $|N(v)| < d$. With this, we reduce the running time for a check for a single neighbor from $O(\Delta)$ to $O(d)$. Accordingly, checking for a vertex whether it is dominated by one of its neighbors is done $O(d^2)$.

Edge Domination [19]
A hyperedge e is dominated by a hyperedge e_1, when $e \subseteq e_1$. e can be removed from the graph.
Proof: See [19]. This yields $I' = I$ and $\alpha(H') = \alpha(H)$.
Complexity:
We check whether an edge e is dominated by an overlapping edge e_1 by checking whether e is a subset of e_1. All edges are sorted in our framework, so this takes $O(min(|e|, |e_1|))$. The number of overlapping edges can grow large very quickly, as it is in $O(n \cdot m)$. We therefore limit the number of overlapping edges we consider to a constant q. We further check whether an edge e is dominated only when $|e| < q$, because the larger the edge, the longer the procedure takes and the less unlikely it is to find an edge that dominates it. With the limit q we reduce the time to check for an edge domination to $O(q^2)$.

4.3.3 Inexact Reduction

The former reductions guarantee that we maintain the ability to find the MIS on the original hypergraph $H = (V, E)$. This reduction might remove vertices that are in all MIS, however, we pick vertices for which that is unlikely and whose absence especially improve the performance of other reductions or other operations in the local search.
We remove p percent of the vertices with the highest neighborhood cardinality. This idea was originally introduced by Dahlum et al. [9] for graphs and we applied it to hypergraphs. The removal results in $I' = I$ and $\alpha(H') \leq \alpha(H)$.
Such vertices are unlikely to be in the solution. For a vertex v, the larger the number of neighbors $|N(v)|$, the larger is the chance of at least one neighbor $n \in N(v)$ being in the solution. They also strongly impact the running time of the local search, as scanning for a $(1, 2)$-swap takes much longer when vertices of high degree are involved.
In the inexact reduction, a vertex is only removed, if it is not currently in the best found solution. Being in the solution after a number of swaps, such a vertex is more likely to be in the solution. We chose the vertex with the next highest $|N(v)|$ instead.

4.4 Online Reductions in the Local Search

4.4.1 Basic Operations

We want to balance the effectivity of reductions and their time overhead.
Reductions are executed in an online manner. The decision to remove a hypergraph element during the iterative local search is final in the process and is not reversed if the solution gets worse. When we set back the independent set in an iteration because its size is smaller than that of the previous iteration, we ignore possibility of the previous solution being smaller than before due to reductions.

To uniformly compare the solution sizes in the local search, even after removals, we use the size of the whole solution on the unreduced hypergraph, including the information about removed vertices.
Intuitively, to get the size of the solution on the original instance $H^O = (V^O, E^O)$, we add the number of included vertices to the size of the current independent set $|I|$ on the current reduced hypergraph $H = (V, E)$. We further need to add the number of foldings executed so far.
Likewise, we update the best solution we found to the current solution only when $|I| + |included| + |foldings|$ is larger than the size of the overall best solution we found on the original hypergraph so far.

In order to avoid checking a vertex for a reduction repeatedly, even when its neighborhood has not been modified, we maintain a reduction-specific marking for each vertex and edge. For a reduction, we maintain a block of marked, unmarked and removed vertices or edges. We only execute the reduction on the element, when it is unmarked. The marker is built in a way that allows us to check for a vertex whether it is marked or to pick a random unmarked vertex within in $O(1)$. We mark vertices and edges that were checked already, but not removed. We unmark a vertex, whenever its neighborhood or incident edges change and an edge, whenever its contained nodes change.

In general, we reduce the hypergraph until all vertices are marked and no further reductions are possible. Especially when working with a preprocessed instance, for which an exhaustive search was already concluded, we make use of the inexact reduction. Beyond the benefits of the inexact reduction for operations within the local search itself, in such cases it is essential to make any further reductions possible. We combine the exact reductions with the inexact one, so that when removal of vertices and edges becomes more seldom and the graph is exhaustively searched for all exact reductions, we extend the number of possible exact reductions by removing vertices that are improbable to be in an MIS. The increased number of elements that can be removed by exact reductions can help find further improvements especially when the local search stagnates without them.

4.4.2 Reduction Placement

We consider three different places within the algorithm where reductions can be incorporated. These affect how often a reduction is executed, which vertices are picked and therefore, which subsequent operations are impacted or skipped entirely in case of removal.

1. Before scanning a vertex for a $(1, 2)$-swap:
Before a scan for a $(1, 2)$-swap, we check for the currently considered vertex and a random incident edge whether it can be removed from the solution. If we can definitely include or exclude a vertex from the solution, there is no need to scan it. We remove it from the hypergraph and continue with another vertex. However, the combined reductions have a high complexity compared to the scan itself, probably outweighing the scan in terms of time and effectiveness of the procedure.

2. Perturbation:
Before forcing a vertex into the solution, we first check whether it or a randomly selected incident edge can be removed from the hypergraph.
If it can be excluded from the solution, it is unlikely to lead to an overall improvement, and we save an entire iteration of the iterative local search. In case it is removed, we force another arbitrary non-solution vertex into the solution. We do not check this one for reductions, in order to not spend too much time on such checks.

3. Iteration:
Before each iteration, we select a random unmarked vertex or edge for a reduction. Vertices and edges from outside the currently considered neighborhood are removed, from different places in the hypergraph. This could lead to more equally distributed reductions in total.

Due to the complicatedness of its concrete implementation in combination with the local search, we consider only the random choice execution before the local search iteration for the vertex folding.
Within the scope of a singular place to execute the reductions, we ordered them with increasing complexity. We save time by considering the simple reductions first and only continuing with the more complex ones if simple ones did not lead to a removal.

4.4.3 Limiting Reductions

We restrict the quantity of reduction calls, as executing them too often could impact the running time of the iterative local search heavily.
We consider setting a frequency, executing a reduction every few iterations with probability $1/f$. Furthermore, we delay the reductions by a number of iterations as in the beginning,

we still find many improvements very quickly and the reductions hinder the process at first through their time overhead.

The inexact reduction is also delayed. In the beginning, excluding a vertex that might be in an MIS could diminish the size of the best approximation we could find on the reduced graph. Instead, we delay it and remove vertices with large neighborhoods only when they are not a part of the solution after a certain number of iterations. With largely differing graph sizes, limiting iterations accounts for longer iterations on larger and more complex hypergraphs.
We execute the inexact reduction either every i iterations for a number i or whenever all vertices and edges were exhaustively checked for all exact reductions.
After stopping the local search, we reconstruct the solution by adding all included vertices and deciding for all folded vertices, whether they are a part of the resulting independent set. This way, we obtain an approximated MIS on the original hypergraph.

4.4.4 Data Structures

For the inexact reduction, we keep a sorted list of all vertices. We sort all vertices once at the beginning and do not update this sorting with the reduction of the hypergraph, the reason being that sorting anew with every change or before every new inexact reduction is too expensive. Additionally, a vertex might have relatively many neighbors on the reduced hypergraph, but not on the original hypergraph. The removal of such a vertex is less likely benefitting the finding of an MIS on the original instance.

To remove a vertex or an edge from an array structure, we first swap it to the end and reduce the arrays' logical size. For blocks of vertices, we add a block of all removed vertices. We reconstruct the whole hypergraph and all related data structures when the total size has reduced by 50 $\%$. For several operations such as the final reconstruction of the solution, we store a mapping between current IDs, resulting from a number of hypergraph reconstructions, and original ones.

As mentioned before, we sort the neighborhoods and edges of the hypergraph, as it strongly accelerates many operations, such as determining, whether two vertices are adjacent or whether an edge is subset of another. When operating on a neighborhood that is sorted by degree, we pick vertices with low-degree first in $(1, 2)$-swaps. These are the most probable solution candidates and also the fastest for consideration. In many cases, a scan does not reach the high degree vertices before finding a low-degree vertex for which a swap is possible.

4.5 Our Addition to the Preprocessing

We apply a reduction preprocessing by reducing a hypergraph with our reductions prior to the local search. We examine the performance of local search with online reductions and with preprocessing as well as their combination.
By choosing some reductions that have not already been used in the preprocessing given by Großmann et al. [19], we extend the space of possible reductions. Thus, we implement an addition to the previous preprocessing. However, we leave out the inexact reduction, as its primary benefit is making more reductions possible after the local search stagnates and speeding up specific operations in the iterative local search and removing vertices beforehand without certainty about their status in an MIS affects the solution space too strongly.
Here, we do not execute reductions in a particular order as done by Großmann et al. [19]. We simply loop through the reductions and check all vertices and edges that are unmarked for the current reduction, using a marker as described earlier. We repeat the loop until all vertices and edges are marked for all reductions.

CHAPTER 5

Experimental Evaluation

In this chapter, we first examine different parameters for our iterative local search (LSHyMIS) and the corresponding local search including online reductions (HyMISOnline). We then evaluate the resulting configuration and the effect of preprocessing on it. Lastly, we compare our algorithm to benchmark local search algorithms on graphs and an exact integer linear program (ILP) on hypergraphs.

5.1 Methodology

The project was implemented using C++17 and it and all comparative algorithms were compiled using gcc 13.3.0 with full optimization flag (-O3) turned on. All experiments were executed on a server with Intel(R) Xeon(R) Silver 4216 CPUs running at 2.10 GHz with 16 cores and two threads per core, possessing 99 GB RAM, sixteen 512 KiB L1d-Cache instances, sixteen 512 KiB L1i-Cache instances, sixteen 16 MiB L2-Cache instances and one 22 MiB L3-Cache instance. For solving the ILP, we used Gurobi [21] version 11.0.3, limiting the process to use a single core. All experiments were executed with GNU parallel, setting the number of jobs to the number of physical cores.

For tuning our algorithm, we run it with a time limit of 10 minutes and run every configuration twice for different seeds, averaging the results. For the evaluation and comparison, we give a time limit of 30 minutes and take the average over four runs with different random seeds. Whenever a graph was preprocessed, the preprocessing time is counted in the time limit.

Instances. We work with a large set consisting of 51 hypergraph instances similar to that used by Großmann et al. [19]. The instances were randomly selected from a set of 488 hypergraphs by Gottesbürgen et al. [16]; an instance from the DAC 2012 Routability-Driven Placement Contest[32] (dac2012), two circuit design [2] (ispd98) instances, 22

instances emanating from general matrices of the SuiteSparse Matrix Collection general matrices [11] (ssmc) and 26 instances originating from the International SAT Competition 2014 [5] (sat14). We list the instances and their properties in Table A.1 of the appendix. For the tuning experiments for the parameters of our algorithms we have a second, small subset of our large instance set. In both sets, we included instances that are easily solved by optimizers and instances that are unsolved within the given time limit. The hypegraphs cover a range of densities, though mostly they are sparse. We also consider hypergraphs with differing levels of reducibility by the reductions we used, as can be observed in the speedups given by Großmann et al. [19] when using their preprocessing compared to utilizing no reductions. The hypergraph sizes stretch from ten thousands to a few million of vertices and edges.
The graphs were given in the format of hMetis [25] and we ignored vertex and edge weights when given, as they are irrelevant for the unweighted MIS problem.
For graph algorithms, we executed the algorithm for the $corresponding\ graph$ acquired through clique expansion of the hypergraph. We subtracted the time for the conversion from the time limit.

Evaluation. We measure the $solution\ quality$ relative to the best solution $bestS$ that was found for an instance over all experiments conducted in the scope of this thesis; the $solution\ quality$ is given by $\frac{S}{bestS}$, where S is the size of the solution found in the current run. For the best solution, we take the largest size of an independent set we found for a hypergraph instance over all runs conducted in the scope of the experiments of this thesis. As running time for the algorithms with a time limit, we contemplate the time it took the algorithm to find its best solution. If the ILP could not solve the instance, this time is the limit of half an hour.
We depict the performance of an algorithm with performance profiles as proposed by Dolanand Moré [12]. In particular, for quality evaluation, we give the percentage of instances for which an algorithm found an independent set with a size at least as large as $\tau \cdot bestS$ subject to the factor $\tau \in [0, 1]$. The larger the percentage of instances for which a solution of $\tau \cdot bestS$ was found for a growing τ, the better the performance of an algorithm. When evaluating time, we instead calculate a factor τ of the shortest time found during all experiments for a single instance. All factors τ are larger than or equal to 1, as we already give a factor of shortest time and a factor $\tau < 1$ implicates a better running time. The more instances the closer to $\tau = 1$, the better the performance of an algorithm.
Especially when a single instance performs particularly good or bad, its impact is not too large in such a profile, making better judgement of the real average performance possible.
For preprocessing, we used preprocessing (PreGSSW) given by Großmann et al. [19] and our own additional preprocessing. We executed these consecutively, starting by applying the PreGSSW preprocessing to an instance and then processing the resulting reduced instance with our own preprocessing producing the $final\ preprocessed\ instance$. The following algorithm was called for this $final\ preprocessed\ instance$.

5.2 Parameter Fine-Tuning

A set of 16 arbitrary instances out of all 51 was used in the following tuning of the parameter of the local search algorithm and the online reductions. The instances are of various rates of reducibility, and sizes, and include instances that are exactly solvable within 30 minutes and instances that are not. For all experiments in this section, we limit the time to 600 seconds and take the average over two runs with different random seeds for each configuration.
Many operations within the iterative local search and especially the integration of the online reductions can be adjusted in order to change their behavior in the algorithm. To test the best version out of several different options for single operations, we introduce a number of parameters which we modify to find the best configuration for each operation.
We try out different heuristics for the local search and the inexact reduction by assessing different sorting criteria for vertices. We further test different places to execute reductions within the local search, achieving different effects on the progressing of the local search. Lastly, especially when executing in an online manner, we need to weigh the time we allow them to consume and their effectiveness, as we wish to maximize the solution size in a given time limit. Therefore, we regard several limiting options that reduce the time within a reduction or the overall number of calls for a reduction.
We first leave out the preprocessing for the tuning of the parameters and compare the effects on previously reduced instances in the following section for the large set of instances.
For the tuning experiments, we start by testing different values for a single parameter. We fix this parameter to the best value found in these tests. When combining, we continue by testing the next parameter in combination with the first parameter set to the fixed value, hence building the global best parameter setting step by step. This way, we aim to balance out the mutual effects of parameter values on each other and improve their impact on the performance. For example, different reductions influence the effectiveness of other reductions, e.g. removing a vertex in the degree one reduction could lead to a vertex folding that was not possible before.

5.2.1 Local Search

Heuristics chosen for a local search impact the algorithms' performance significantly. We compare different sorting criteria according to which to sort the neighborhood of each vertex. We compare the relative solution quality for the different criteria, the results are shown in Figure 5.1. We sort the neighborhood either by the degree, neighborhood cardinality or ID of a vertex or choose a random key to sort it by. Figure 5.1 shows that sorting by the degree of a vertex is better by over 0.01 $\%$ than the alternative criteria. This is because during a scan for a 2-improvement, a vertex with the lowest degree has the highest chances to be involved in a $(1, 2)$-swap. It is the least likely to be connected to other free vertices in the neighborhood.

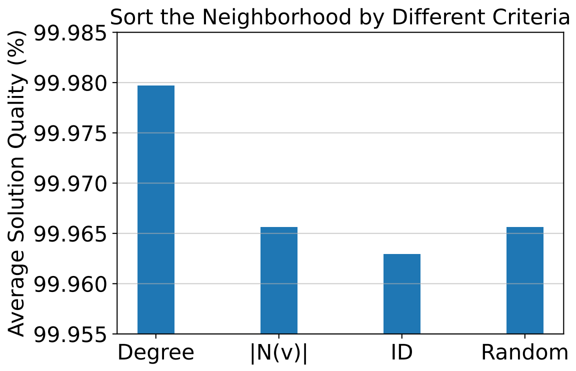


Figure 5.1: Sort the neighborhood by either the degree, the cardinality of the neighborhood or the ID of a vertex or a random key

5.2.2 Separate Reduction Assessment

When incorporating reductions into the local search, there are different possibilities of where to apply them. In this section, we first test for the low-degree-vertex and low-size-edges reduction as well as edge and vertex domination, where to apply them in the iterative local search. We assess three options:

1. Executing them for a vertex v before scanning v for a $(1, 2)$-swap in the local search step.
2. Checking a vertex for reductions before forcing it into the solution in the perturbation.
3. Executing each reduction for a random unmarked vertex or edge at the beginning of an iteration.

For edge reductions, when executing it before forcing a vertex into the solution or scanning it for 2-improvements, we pick a random incident edge of this vertex.

The inexact reduction and vertex folding are always conducted at the beginning of an iteration. The inexact reduction is not specific to any vertex, we measure different parameters for it. For simplicity, due to the complicated nature of the implementation of the vertex folding during the iteration of the local search, we always execute it for a random unmarked vertex at the beginning of the iteration.

Furthermore, some reductions consume much time for certain types of edges and vertices. We test limits within the reductions that are meant to constrain unnecessarily complex cases. For the limit tests, we fix the place for the reduction.

When regarding a single reduction, we always test it in combination with the inexact reduction, to estimate the true effect when combining the reductions later on. By removing vertices and edges, a reduction can enable another reduction to succeed in removing a vertex that could previously not be removed by it.

A reduction alone might stagnate quickly, so we apply the inexact reduction every 100 thousand iterations. We note that the overall qualities are lower here than in later experiments, due to the frequent call of the inexact reduction, but we consider only the ratio of

different parameter values to determine a good configuration.
Figure 5.2 gives an overview over the resulting solution qualities for all parameters for each exact reduction. The solution size after a short time limit of 10 minutes indicates how well-balanced the time spent on reductions and the gain achieved by their execution is.

Low-Degree Vertices
The degree zero, degree one, zero neighbors and one neighbor reductions are executed in one common simple check of the vertex properties.
We compare the different options of places in the local search to execute the reductions. We can see clearly in Figure 5.2 that executing low-degree node reductions before the local search yields much higher solution qualities than the other options.
The reason for this is that forcing such a vertex into the solution and conducting the whole subsequent iteration can take a lot of time. For example, we consider a vertex v has degree one and many neighbors. When forcing v into the solution, there is no possible scan in the neighborhood, but the local search might take long. Especially when finding no swaps, a scan considers all neighbors without breaking off earlier, making this step especially time consuming.
Executing it before the perturbation, we now test different limits. This is important especially for sparse instances with many large edges. When including a lot of one-degree vertices with many neighbors, one removal takes longer than for smaller neighborhoods, as all neighbors have to be excluded from the solution first. When this happens frequently, it consumes a significant amount of time in total.

Low-Size Edges
For the size zero and size one edges reduction, we execute them in one step as a simple check of the edge size. We consider the different places within the iterative local search to execute this check. We depict the relative solution qualities in Figure 5.2.
It can be observed that the local seach performs worst when executing it in every scan. This is, because the scan for $(1, 2)$-swaps is more frequently called than the alternatives and executing a reduction too often takes away too much time from the local search itself. Due to the similar solution sizes when performing the low-size edge reductions before the perturbation or before an iteration, we pick the perturbation to execute the reductions. Subsequent vertex reductions in the same neighborhood, instead of reducing random vertices, have an increased chance of being successful and could prevent an unnecessary local search step. Furthermore, in case of the edge domination also being executed in the perturbation, this speeds up the process. As a small edge might still have many overlapping edges, possibly removing it before executing the edge domination is also beneficial.

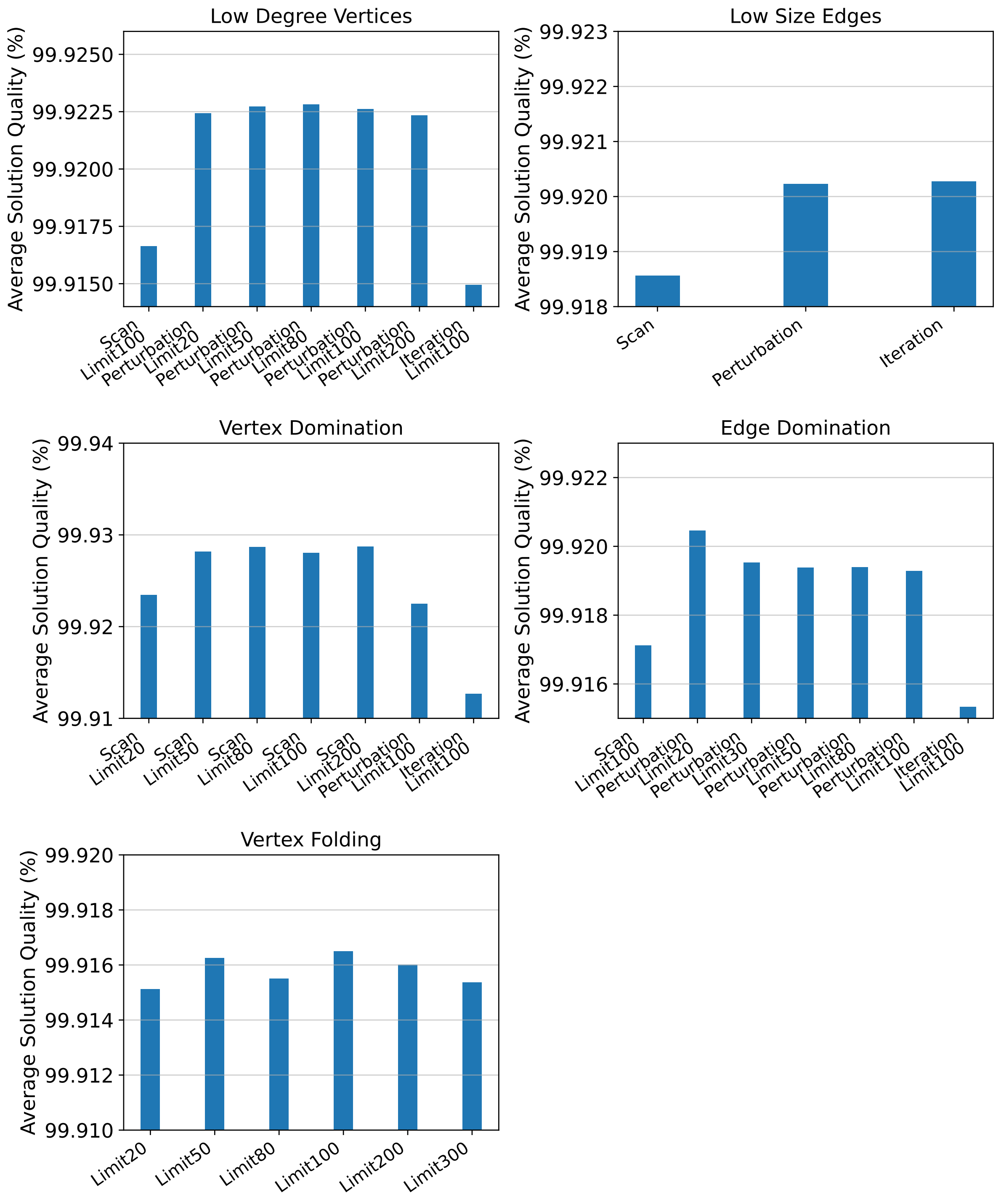


Figure 5.2: Place and limits for exact reductions. For each reduction we give the relative solution quality when executing it before scanning a vertex (Scan), before forcing a vertex into the solution (Perturbation) or for a random vertex before the iteration (Iteration). For the best place for a reduction, we give solution qualities for different limits. We limit $|N|$ for vertex reductions and $|e|$ for edge domination.

Vertex Domination
We first test the different places to apply the vertex domination. We can see in Figure 5.2 that the local search finds the largest solution qualities when checking if a vertex dominates another before scanning it for 2-improvements. The complexity of the vertex domination reduction and the scan are similar, both operate on the neighborhoods of a vertex and their neighbors. The improved solution quality is possibly memory-related. Operating on the same vertices in the reduction can lead to fewer cache misses in the subsequent scan for a $(1, 2)$-swap. Furthermore, a successful vertex domination excludes a vertex from the solution. All vertices that have become free during this operation are iteratively inserted into the solution. In case the subsequent scan had led to a swap, the vertex domination probably leads to an improvement of the solution size.
As for each neighbor of a vertex v, checking whether v dominates it is in $O(|N(v)|)$, we limit the neighborhood cardinality for which we check a vertex. Figure 5.2 shows that limits of 80 or 200 neighbors for vertex domination before the scan are particularly good. We choose a limit of 80, as larger values could easily become a bottleneck for highly connected graphs with large neighborhoods.

Edge Domination
We compare solution qualities for executing the edge domination in the different places. Highest solution qualities are observed when applying it before the perturbation. Choosing edges within a specific neighborhood, it is more likely that following vertex reductions are successful in removing a vertex, which is why it is much better than picking a random vertex at the beginning of the iteration.
The scan is worse as a place for this reduction than the perturbation, because of the high number of calls in the scan. Also, the time we save by not executing a scan is shorter than the time we save by not executing a whole iteration in case the vertex to which the edge is incident is removed before the perturbation.
The number of overlapping edges for an edge e is in $O(n \cdot m)$. Checking for every overlapping edge whether e is dominated by it is much too complex. We limit $|e|$ and the number of overlapping edges we consider by a constant value.
We test different limits for edge domination before the perturbation. Figure 5.2 shows that a limit of 20 is best. Edge domination consumes much time but only has an indirect effect, thus a low limit of 20 leads to the largest average solution quality. Also, smaller edges are more likely to be dominated, so by setting a small limit, the reduction is faster and more likely to succeed in removing an edge.

Vertex Folding
In Figure 5.2, we depict different limits and the resulting solution qualities of the local search for a constant value limit. As we can see, a limit of about 100 yields the largest solution qualities. Checking whether a vertex and its neighbors can be folded is done quickly. However, the process of folding itself takes longer with increasing neighborhood size. Also, folding vertices produces larger neighborhoods, in turn also negatively im-

pacting the performance of scans for 2-improvements. To weigh the positive and negative effects, we limit $|N(v)|$ for a vertex v.

Resulting Places and Limits: We continue for the vertex domination with a limit of at most 80 neighbors in the scan. In the perturbation, we execute the low-size edge reduction, the edge domination with a limit of at most 20 contained edges and the low-degree node reduction limiting the number of neighbors to 80. The vertex folding reduction is executed for a limit of 100 neighbors for a vertex in the following. Overall, the majority of reductions had the most positive impact on the solution quality in the perturbation. Being less frequently called, the time spent on the reductions there is smaller than calling them for every scan. Reducing the neighborhood before the local search step furthermore prevents a larger number of operations, such as multiple scans in the local search step, than reducing before the scan, which only prevents one scan for $(1, 2)$-swaps. We now run the local search applying all reductions using these best configurations for each.

Inexact Reduction

The inexact reduction takes longer than the others on average, as we first need to sort all vertices. Also, it might remove vertices that are in every MIS, thus decreasing the possible best solution that can be found on the reduced instance. The risk of that happening must be kept small. To achieve the best possible results, we tune different parameters for this reduction by applying it in combination with all other reductions.

Percentage. We first run the local search, executing the inexact reduction every four hundred thousand iterations, and set the percentage of vertices we remove from the hypergraph to different factor values.

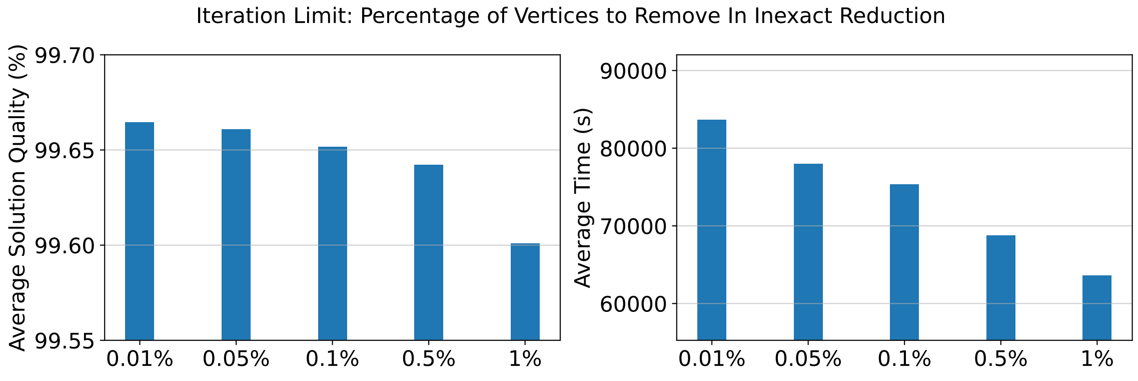
Initially we got much worse solution results when applying the inexact reduction more often, which led us to regard an iteration limit instead of a time limit for tuning this parameter. We did this in order to find out whether removing a percentage the vertices inexactly is overall disadvantageous or yields worse solution qualities due to the time overhead.

We limit the iterations to $1.9 \cdot 10^6$, which is roughly the lowest number of iterations an instance out of our set of 16 hypergraphs managed within 10 minutes for previous experiments.

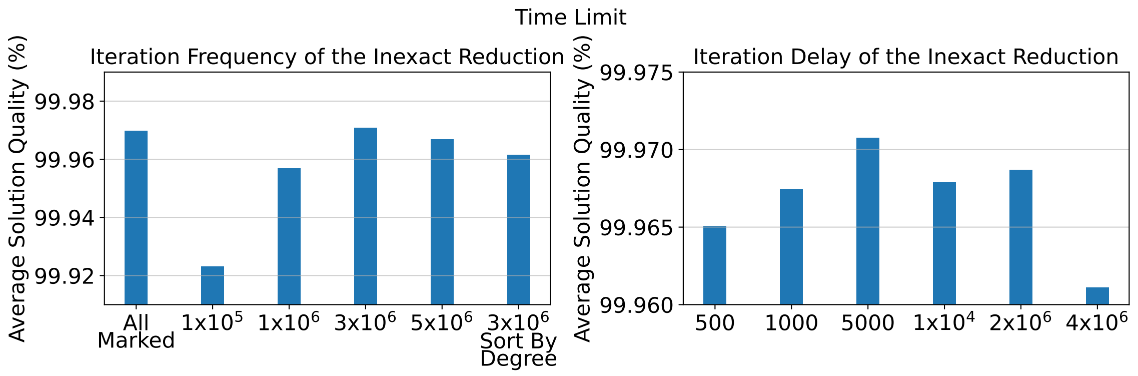
In Figure 5.3, we can see that the relative solution quality indeed increases for lower percentages, 0.01% leading to the best results.

Removing fewer vertices here suffices, as removing vertices with large neighborhoods impacts many neighbors and can enable many exact reductions on the neighborhood and in turn leads to more possible reductions stretching over the whole graph.

Furthermore, we remove vertices with the highest count of neighbors, thus already limiting the number the vertices that lead to the largest bottlenecks in the algorithm.



(a) We give the relative solution quality for different percentages to remove in the inexact reduction and the corresponding time when limiting iterations.



(b) We give the relative solution qualities found within 600 s for delaying the reduction by a number of iterations. Solution qualities are also given for runs where we execute the inexact reduction either every k iterations, k being the frequency parameter, or when all vertices and edges are marked for all exact reductions.

Figure 5.3: Parameters for the inexact reduction

With online reductions, the balancing of time and effectiveness of a reduction is an especially delicate matter. Here, we choose 0.05% to remove in every inexact reduction. The corresponding solution quality is the second highest, while also leading to a much reduced time overall, as depicted in Figure 5.3(a).

Frequency and Delay of the Inexact Reduction. Again limiting the local search with a time limit of 10 minutes, we now observe different frequencies and delays for the inexact reduction. As times for a single iteration differ greatly, we choose a number of iterations for the frequency and delay of the inexact reduction. The test results for different frequency parameters are shown in Figure 5.3. We have one special case, where we execute the inexact reduction whenever the exact reductions have exhaustively checked the hypergraoh and all vertices and edges are marked. Executing the inexact reduction every $3 \cdot 10^6$ iterations leads to the best results. We do not need to perform this reduction

often. The chance of it removing vertices that are definitely in an MIS grows too large otherwise. However, sometimes removing vertices of high degrees increases the speed of other reductions whose running time is dependent on $|N(v)|$, $deg(v)$ or $|e|$ for a vertex v and an edge e and increases the number of possible exact reductions.
In Figure 5.3, we show that delaying the inexact reduction improves the solution quality of the local search. A delay of 5 000 iteration leads to particularly high qualities. Delaying the inexact reduction balances out its negative effect. After a number of iterations, high degree vertices that are in the solution are likely to be in an MIS and those that are not, are more likely not to be in an MIS than non-solution vertices were in the beginning, before executing the local search. We do not remove the vertices that are currently in the solution and skip them instead.
In the following, the inexact reduction it called every $3 \cdot 10^6$ iterations with a delay of $5\,000$ iterations. It removes 0.05% of all vertices each time. We compare for the best configuration for previously tested parameters the results when sorting all vertices by the count of their neighbors or their degree. We can see that sorting by the count of neighbors is superior to sorting by the degree in approximating MIS. The reason for this is that some operations, in particular the $(1, 2)$-swap, have a time dependent on $|N(v)|$ for a vertex v. Moreover, the number of vertices affected by the removal of the vertex is probably larger, aiding more exact reductions.

Overall Frequency and Delay. Figure 5.4, first depicts the average solution quality when executing all reductions only every k iterations for a constant number of iterations k. The results show no clear tendency, peaking for several parameter values. These peaks happen when a single graph achieves its best average solution for a specific value. For one, barely reducible, instance, the impact is especially large. The quality of the solution found for it grows with fewer reductions while it slowly diminishes for many others. This leads to peak at 100 for the average. These results vary strongly, giving no clear best parameter value. As executing all reductions every time is the otherwise best configuration besides 100, we use $k = 1$.

Lastly, fixing all previously best parameter values, we now run the local search for different delays for all reductions. The results depicted in Figure 5.4 indicate a delay of $6 \cdot 10^6$ is best for the reductions. For many hypergraph instances, hundreds of millions of iterations are conducted within 10 minutes. In comparison, the limit is relatively small. For all instances, not slowing down the local search in the beginning, when we still find numerous 2-improvements in a short time, proves advantageous.

Reductions are delayed by $6 \cdot 10^6$ iterations and executed in every one after that. We have analysed different parameter values and evaluated the corresponding solution quality. This way, we have achieved a good configuration for our experiments. We note that best versions for some parameters change significantly for different instances, and it might be profitable to estimate rough values for parameters for other instance sets by

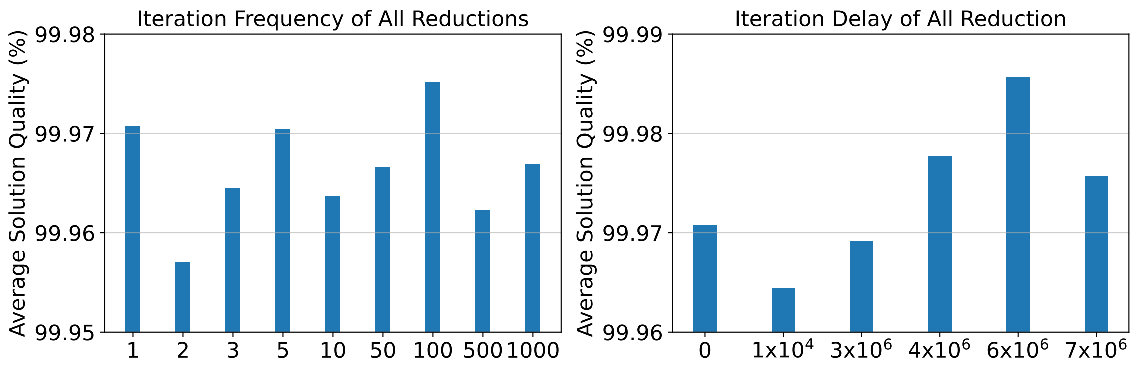


Figure 5.4: Frequency and Delay for all reductions. We give the number of iterations in between executing reductions and the number of iterations by which we delay them.

running a small tuning before using this project. These parameters include the frequency for the inexact reduction and the frequency and delay for all reductions.

5.3 Evaluation of the Best Configuration

We assess our iterative local search for the best configurations found during the fine-tuning experiments, considering the version with and the version without reductions. We compare the performance when applying no preprocessing and when operating on preprocessed instances. The large set of 51 hypergraph instances is used in the following and we take the average over four runs with different random seeds.

5.3.1 Reductions Values

We first compare the average count of $elements$ (vertices or edges) removed by each reduction as well as the time spent on the reduction overall for our algorithm when using no preprocessing and when operating on a preprocessed hypergraph.
For the version without preprocessing in Figure 5.5(a), edge reductions remove more elements each than a single vertex reduction. As we consider a larger number of vertex reductions, though, the overall number of removed vertices is larger than the number of removed edges. A single vertex reduction removes fewer vertices due to other reductions removing the vertex first. This is part of the reason why low-degree vertex reductions take the most time compared to the count of vertices removed by it. As mentioned earlier, including a vertex into the solution also takes longer, as we need to exclude all neighbors beforehand. The manner of our implementation plays a part in the time overhead, as we could implement a combined removal for example for all vertices within an edge, instead

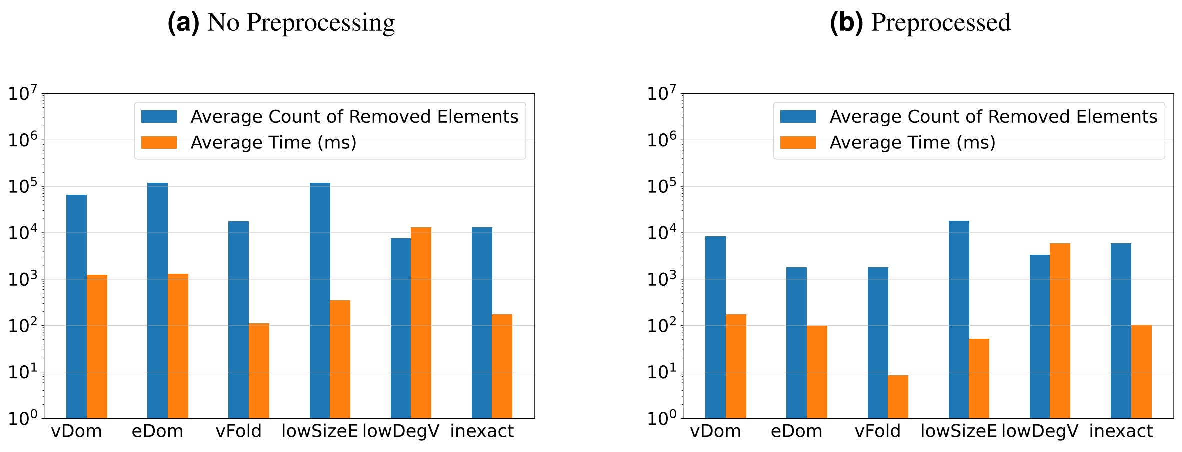


Figure 5.5: Number Of Vertices and Edges Removed for each Online Reduction

of updating every removal separately.
These relations remain roughly the same after preprocessing, as depicted in Figure 5.5(b). For an instance that was reduced beforehand, we find fewer additional reductions overall. The inexact reduction proves successful in enabling the others to further reduce the graph instance, even after a previous exhaustive reduction of the hypergraph. With a smaller number of iterations, the inexact reduction was called fewer times as well, leading to fewer removed vertices for this reduction on average. We can see that the number of elements removed by the more complex reductions that were already conducted in the preprocessing, go back especially, impacting the edge reduction the most. Even though the number of elements removed during vertex folding decreases, the reduction becomes much faster, as the number of vertices and the edge sizes are smaller in preprocessed instances (see Table 5.1).

Table 5.1: Hypergraph sizes for different reduction configurations. PreGSSW is given by Großmann et al. [19]. We further give sizes after the total preprocessing, adding our own after PreGSSW, sizes after the local search with only the online reductions and lastly after the combination of preprocessing and online reductions.

	n	m	$\lvert H\rvert$	$\lvert e\rvert$	$t(s)$
original	327 185	681 021	2 791 030	17.92	0.00
PreGSSW	207 625	312 224	1 030 066	8.88	6.91
total preprocessing	183 227	232 845	858 376	8.96	7.40
online	217 500	444 952	1 547 748	15.30	3.59
online&preprocessing	159 472	210 087	682 399	8.41	7.89

We now compare for a hypergraph $H = (V, E)$ the number of vertices $n = |V|$, the number of edges $m = |E|$, the hypergraph size $|H|$, which is the sum over all edges sizes, the average edge size $|e|$ and the total time spent on reductions. In Table 5.1 we give the average hypergraph properties over all 51 instances of the large hypergraph set for different combinations of all reduction algorithms.

The PreGSSW preprocessing given by Großmann et al. [19] reduces a hypergraph to roughly a third of its original size $|H|$ on average, the mean edge size shrank by over 50 $\%$ in about 7 seconds. Adding our own preprocessing further decreases the overall hypergraph size $|H|$ to under a third of the original instance in less than a second more. The edge size, however, is not reduced further.

Using only online reductions, the average reduced instance after the local search is more than half as large as the original hypergraph, and much larger than the reduced hypergraph gained by the total preprocessing. However, we spend half as much time on these reductions on average as we do on the preprocessing. We implemented the algorithm with the goal to reduce the overhead of reductions and still find high quality solutions in a fast manner, aiming to spend less time on reductions in between local search operations overall. Instead, more specific reductions over the time are meant to improve the solution, thus, the algorithm takes less time for the online reductions than the preprocessing takes.

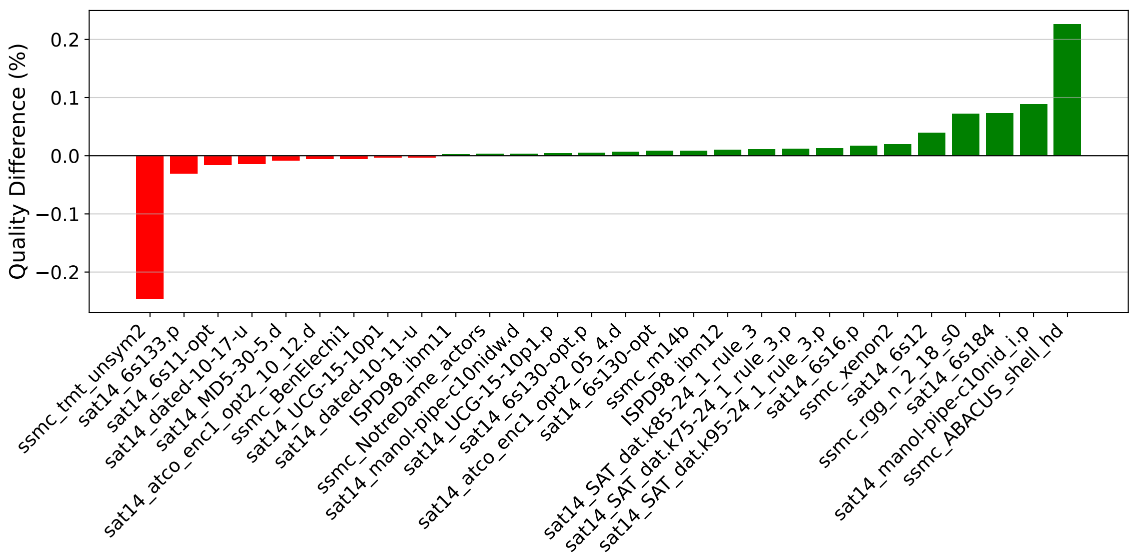
The hypergraph size $|H|$ becomes smaller the more reductions are applied. The preprocessing removes the most vertices and edges, however the following online reductions decrease the size of the hypergraph further, again in less than a second more than the total preprocessing. On average, an instance was reduced to a quarter of its original size $|H|$ in roughly eight seconds in total when using all possible reductions.

We now give the difference of relative solution qualities of the local search with and without online reductions in Figure 5.6. A difference larger than zero implicates that the application of online reductions leads to an improvement compared to not utilizing them. In Figure 5.6(a), we compare the performance on instances that were not preprocessed. We see an improvement larger than 0.002% of the relative solution quality for 19 instances. For 9 instances however, the relative solution quality is worse when using online reductions by over 0.002%.

When applying the preprocessing, as shown in Figure 5.6(b), the solution quality is larger for only 7 out of 51 hypergraphs, getting worse for 11 instances. Still, the algorithm performs significantly worse on only one instance (ssmc_tmt_unsym2) that is barely reducible by the reductions we applied, as the reduced size is barely smaller than the size of the original instance.

On average, the utilization of online reductions improves the solution quality when working on hypergraphs that were not preprocessed. On preprocessed instances, the overall performance improves when excluding the instance that is not reducible by our reductions. This is because finding further reductions of the same type as those used in the preprocessing is harder after an exhaustive reduction has already been conducted.

(a) No Preprocessing



(b) Preprocessed

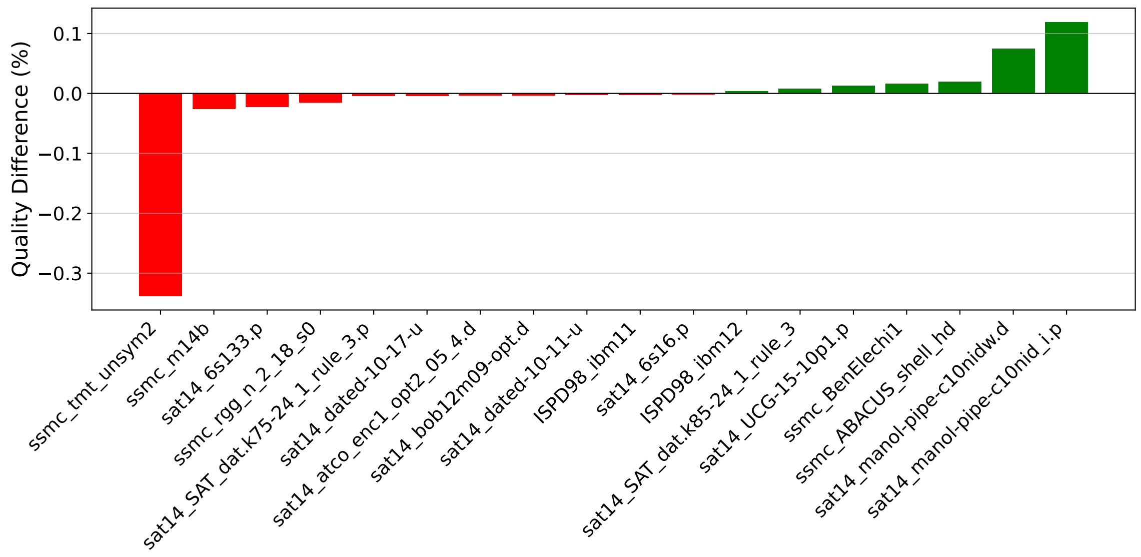


Figure 5.6: Quality difference. For each hypergraph, we calculate the relative solution quality $\frac{|I|}{bestS}$ of the solution for the algorithm with online reductions $|I|$. We substract the relative solution quality $\frac{|I^O|}{bestS}$ of the solution I^O found by the local search without online reductions and give the resulting difference in $\%$. Only instances with a quality difference of more than 0.002 $\%$ are portrayed.

5.4 Comparison

In this section, we compare our iterative local search (LSHyMIS) and the iterative local search including online reductions (HyMISOnline) to two other local search algorithms on graphs and an ILP on hypergraphs provided by Großmann et al. [19]. The local search algorithms on graphs are another framework using online reductions by Dahlum et al. [9] (OnlineMIS) and a local search for the maximum weighted independent set problem introduced by Großmann et al. [18] (CHILS). For CHILS, we use the baseline local search of the framework. The maximum weighted independent set problem is equivalent to the MIS problem for all weights being one. We give weights of one for each edge and vertex for the solving of the weighted problem. We further convert the hypergraph to a graph using clique expansion to compare the graph and hypergraph algorithms. We compare their performance with and without the application of the preprocessing (PreGSSW [19] & our own preprocessing addition). The average results for each instance for all compared algorithms in combination with the preprocessing are given in table A.2 and A.3.

Performance Solution Quality
First, we regard a performance profile (Figure 5.7) comparing the solution qualities for each algorithm. Most importantly, even though LSHyMIS reaches 100% of instances a little earlier, the line for LSHyMIS is below that of HyMISOnline most of the time for the version without preprocessing, indicating that we found larger solutions for fewer instances in LSHyMIS. When preprocessing, LSHyMIS again reaches 100% earlier than HyMISOnline. Until a factor of roughly 0.9995, HyMISOnline found a higher percentage of instances for which this solution quality was reached.
Overall, the percentage of instances drops later for all algorithms when operating on preprocessed instances. Especially for working with graphs, the conversion and the resulting

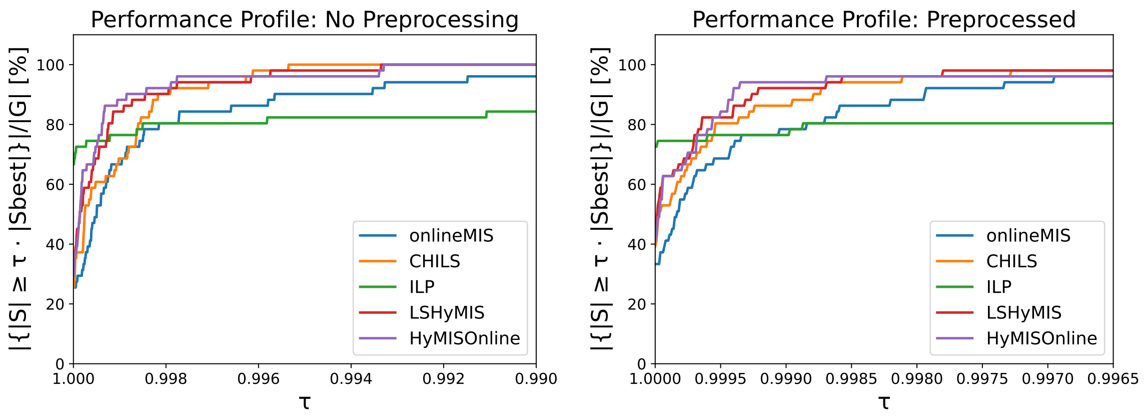


Figure 5.7: Performance profile comparing the relative solution quality of all algorithms with and without preprocessing

correspondig graph is more simple when working with reduced hypergraphs. The lower edge size yields a structure closer to a graph, leading to especially good overall results for CHILS ([18]). The ILP [19] consistently finds good and optimal results for just below 80% of the instances, performing much worse for the residual 20% of large hypergraphs.
In general, we can see that reductions improve the solution quality – online reductions as well as preprocessing. The HyMISOnline algorithm mostly performs better than the LSHyMIS algorithm quality-wise with and without preprocessing.
A factor that could explain the difference when operating on a preprocessed hypergraph is the tuning of the algorithm. As the preprocessing largely impacts the reducibility of a hypergraph instance, more specific tuning might further improve the performance. Another aspect is the similarity of the online and preprocessing reductions. The results might be different for a preprocessing algorithm using less similar reductions.

Performance Time

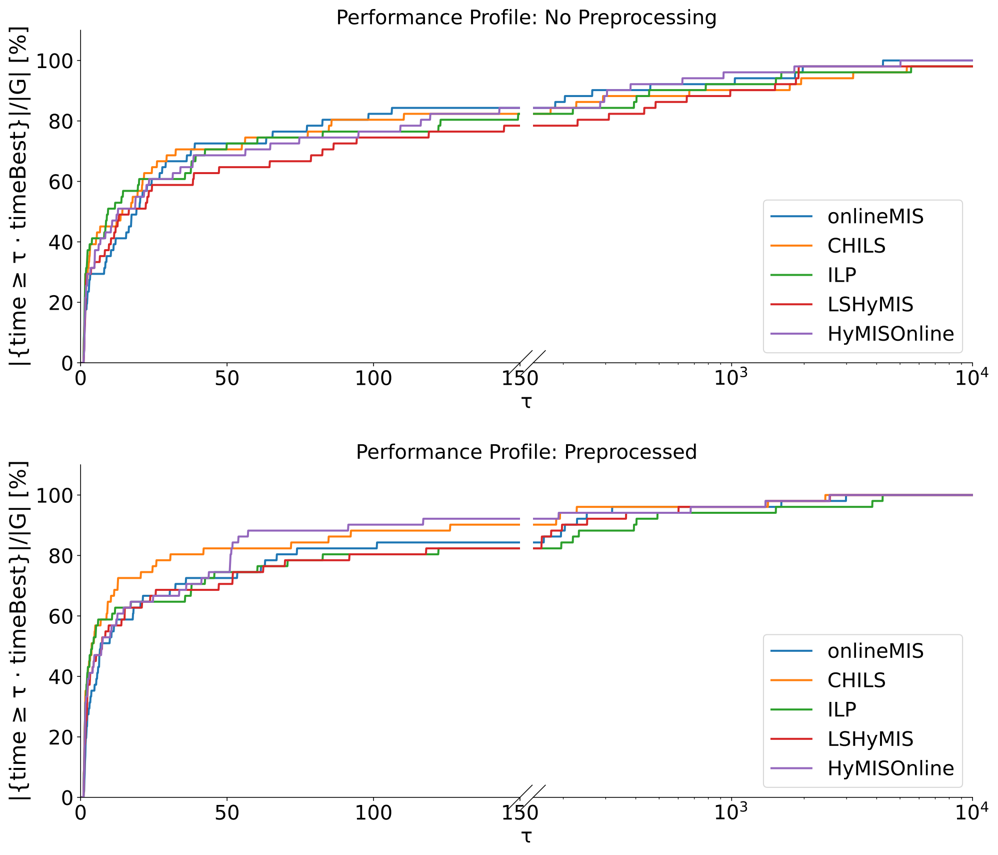


Figure 5.8: Performance profile comparing the factor of the best overall time to find the best solution of all algorithms with and without preprocessing

In Figure 5.8, we compare the running time of the algorithms. As the time for an algorithm, we define the average time until the best solution found during the run was reached. We give the percentage of instances finding the best solution within a time of a factor $\tau > 1$ of the best time we found over all runs for all algorithms. τ cannot be smaller than one, as we pick a factor of the best found time, so all others must be larger.
As depicted in Figure 5.8, when we are not preprocessing the hypergraph instances beforehand, HyMISOnline is always faster than LSHyMIS. When preprocessing that is mostly the case as well, the percentage of hypergraphs for HyMISOnline only dropping below that of LSHyMIS for factors lower than 50. Applying specific reductions before the operations of the iterative local search as described in the fine-tuning speeds up the process.
While the the percentage of instances for which the time was shorter than a factor of the best time for LSHyMIS is often the lowest out of all, HyMISOnline can compete with the graph algorithms.
We note that the conversion from hypergraphs to graphs plays a part in the running time of the graph algorithms. This is especially clear when examining the running time for CHILS; operating on a graph derived from a hypergraph with smaller edges, which is closer to the structure of graphs, it is significantly faster than the other algorithms. Here, again until a factor of about 50, HyMISOnline has a similar performance. In combination with the preprocessing, all algorithms are much faster, a large percentage of instances reaching smaller factors of the best time.

5.4.1 Average Results

When calculating the average relative solution qualities and times, one specific hypergraph heavily impacts the outcome. For the instance ssmc_tmt_unsym2 of the SuiteSparse Matrix Collection [11], we continuously observe particularly bad results when using reductions. As demonstrated in Table 5.2, the graph is barely reducible by all reductions considered in this thesis. The table shows that the hypergraph is reduced to 90% of its original size $|H|$.

Table 5.2: Properties of the original hypergraph of the ssmc_tmt_unsym2 instance and the reduced instance obtained by executing all reductions.

	n	m	$\|H\|$	$\|e\|$	$t(s)$
original	917 825	917 825	4 584 801	5.0	0.0
online&preprocessing	825 046	883 720	4 088 081	4.63	2.66

To analyze the effect of reductions on the overall performance of an algorithm, considering performance on instances that are barely reducible by our reductions is also important for determining their overall effect. In the tuning experiments, this is useful in tuning experiments to balance the performance for such instances as well. However, to draw a sensible conclusion for the average graph instance and the effect of vertex removals, we consider the graph set without the edge case hypergraph. In Table 5.3, we compare the relative solution quality and time averages over all instances with and without the ssmc_tmt_unsym2

Table 5.3: Average relative solution quality q and time t until the best solution was found for all algorithms with and without the barely reducible instance. The notion Pre indicates the application of prepprocessing.

	ssmc_tmt_unsym2		no ssmc_tmt_unsym2	
	$q(\%)$	$t(s)$	$q(\%)$	$t(s)$
OnlineMIS	99.794 1	800.684 3	99.860 7	780.749 5
OnlineMISPre	99.873 2	639.521 5	99.940 3	616.366 3
CHILS	99.920 6	821.368 6	99.924 9	801.960 6
CHILSPre	**99.960 3**	568.206 1	**99.963 3**	543.677
ILP	97.787 2	838.442 6	98.040 2	819.201 5
ILPPre	97.870 8	822.271 5	98.125 4	802.704 6
LSHyMIS	99.937	743.553 1	99.944 2	722.512
LSHyMISPre	99.948 1	640.324 3	99.951 4	617.218 6
HyMISOnline	99.942 7	618.115	99.955	594.655 3
HyMISOnlinePre	99.944 7	**542.915 6**	99.954 8	**517.948 4**

hypergraph instance. The suffix Pre indicates the algorithm was executed in combination with a previous preprocessing.
The performance profile is a better tool to find the best configuration for the average hypergraph, as every instance holds the same weight there, no matter how strongly affected its performance is. For the average presented in Table 5.3, however, HyMISOnline exhibits a solution worse than the quality of LSHyMIS when preprocessing was applied. This is because on the ssmc_tmt_unsym2, more time is spent on reductions unnecessarily. With few successful reductions, we have little positive effect. For example the vertex domination breaks off when the vertex can be removed. If that is rarely ever the case, the reduction takes longer on average as well, leading to more time for less gain, yielding a much worse performance.
Overall, CHILS [18] in combination with preprocessing achieves the best results for solution quality. They find solution sizes of over 99.96% of the best solution in both cases, with and without ssmc_tmt_unsym2. Only HyMISOnline is faster by more than 20 seconds in finding its best solution both times. Without the ssmc_tmt_unsym2 hypergraph, online reductions increase the relative solution quality by 0.01% without and 0.003% with preprocessing on average. However, the best solution is reached around a hundred seconds faster in all cases for HyMISOnline than for LSHyMIS. Applying fewer reductions at carefully chosen places during the local search has a positive effect on the quality of approximations of MIS for the other 50 hypergraph instances on average.

CHAPTER 6

Discussion

6.1 Conclusion

In this work, we applied an iterative local search for the MIS problem to hypergraphs and built a framework that includes reductions in an online manner. For the reductions, we utilized relatively simple hypergraph reductions. We considered low-degree vertex and low-size edge reductions, and the more complex edge and vertex domination. We further applied an inexact and a vertex folding reduction to hypergraphs. Edge reductions lead to a particularly large number of removed elements. Edge and vertex domination perform especially well during the local search when the instance is not preprocessed. Only the low-degree vertex reduction takes much more time than the other reductions in comparison to the number of vertices removed by it. For the online application of reductions, the inexact reduction can enable many more removals on preprocessed instances.
By applying online reductions, we improved the relative solution quality by more than 0.002% on 37% of the hypergraph instances compared to using no reductions. We speed up the finding of the best solution by roughly a hundred seconds. When first conducting a preprocessing, we reduced the average instance obtained by it by 21%. We added a subsequent preprocessing using the reductions we chose for online application except the inexact. Only the vertex folding was not already used in the earlier preprocessing. Our additional preprocessing still decreases the graph size by another 7% of the original graph size. Applying the vertex folding reduction effectively allows other reduction rules to remove more vertices and edges.
We spend half as much time on online reductions when using no preprocessing than we do on the preprocessing itself. However, the lower number of reductions does not lead to worse results for the local search, as the specific application of reduction rules has similar advantages when considering a neighborhood, but overall takes less time.
Lastly, our algorithm can compete with other local search algorithms on graphs. Our local search with online reductions is on average the fastest out of all the algorithms of the comparison experiments. Here, we consider the time it took to reach the best solution found

within a run. The average solution quality for the local search with online reductions is second only to the one other algorithm in combination with preprocessing.

6.2 Future Work

There are many possibilities for adaptation in the algorithm implemented in this thesis. We already introduced various parameters. Options for the inexact reduction are especially difficult to configure. A more detailed evaluation of this reduction could largely improve the performance. We already skip vertices that are in the solution. Similar constraints dependent on other properties of the hypergraph and solution might prove useful as well, possibly even reversing removals that lead to no improvement in a branch-and-bound like fashion. Generally, parameters applied to all reductions at once for simplicity in this project could be tuned reduction specifically. Picking hypergraph-specific values for multiple other parameters, such as the frequency of reductions could lead to further improvement. Here, not picking constant values should give clearer results.
Different combinations of reductions and their effects on each other could also be more thoroughly investigated. On one hand, the results when leaving out single reductions might be interesting. For example, one could leave out or replace the edge domination, which is useful but very time-consuming with growing limits. On the other hand, trying out different reductions, more complex ones, might lead to better results. Especially when combined with the preprocessing, not using similar reductions could impact the performance. When limiting correctly and choosing the best application within the local search could be advantageous. Further optimization of specific implementation details could also lead to better performances in the future.
Another idea is to choose vertices in the scan for 2-improvements not by sorting the neighborhood, but instead picking them with a heuristic dependent on e.g. the degree of a vertex. We do not update the neighborhood sorting when removing a vertex, so our heuristic stays the same even when reducing the hypergraph. A heuristic function could dynamically change for varying graph properties. Other algorithms might furthermore benefit from online reductions as well. We suggest trying out online reductions for different local search algorithms.

APPENDIX A

Appendix

Table A.1: Hypergraph properties for the large set consisting of 51 instances. We give the number of vertices n, the number of edges m, and the average hyperedge size $|e|$.

	n	m	$\|e\|$
ssmc_ABACUS_shell_hd	23 412	23 412	9.33
ssmc_BenElechi1	245 874	245 874	53.48
ssmc_bips07_1998	15 066	15 066	4.13
ssmc_ca-CondMat	23 133	23 133	8.08
dac2012_superblue2	1 010 321	990 899	3.26
ssmc_dictionary28	52 652	39 327	4.53
ssmc_ex19	12 005	12 005	21.65
ssmc_fd18	16 428	16 428	3.86
ssmc_g7jac040sc	11 790	11 790	9.73
ssmc_garon2	13 535	13 535	28.86
ISPD98_ibm11	70 558	81 454	3.45
ISPD98_ibm12	71 076	77 240	4.11
ssmc_lhr14	14 270	14 270	21.57
ssmc_lp_pds_20	108 175	33 798	6.88
ssmc_m14b	214 765	214 765	15.64
ssmc_msc10848	10 848	10 848	113.36
ssmc_NotreDame_actors	127 823	383 640	3.83
ssmc_pdb1HYS	36 417	36 417	119.31
ssmc_poli3	16 955	16 955	2.23
ssmc_psse2	11 028	28 634	4.03
ssmc_rgg_n_2_18_s0	262 144	262 141	11.8
sat14_6s11-opt	66 552	97 312	2.33
sat14_6s12	68 066	99 580	2.33
sat14_6s130-opt	98 654	144 361	2.33
sat14_6s130-opt.p	49 327	144 361	2.33
sat14_6s133.p	48 215	140 968	2.33
sat14_6s16.p	31 483	91 888	2.33
sat14_6s184	66 730	97 516	2.33
sat14_ACG-20-10p1.d	1 632 906	381 708	10.06
sat14_AProVE07-27.d	29 194	7 729	9.98
sat14_atco_enc1_opt2_05_4.d	386 163	14 636	112.93
sat14_atco_enc1_opt2_10_12.d	147 853	9 495	65.15
sat14_bob12m09-opt.d	152 446	51 144	6.95
sat14_dated-10-11-u	283 720	629 461	2.27
sat14_dated-10-17-u	459 088	1 070 757	2.31
sat14_E02F22.p	13 574	1 301 188	8.81
sat14_manol-pipe-c10nid_i.p	252 516	750 877	2.33
sat14_manol-pipe-c10nidw.d	1 291 714	433 601	6.95
sat14_MD5-30-5.d	68 103	8 905	29.88
sat14_openstacks-p30_3.085-SAT	643 132	1 643 601	2.38
sat14_q_query_3_L100_coli.sat.p	315 617	1 522 583	2.89
sat14_q_query_3_L150_coli.sat	973 984	2 456 708	2.91
sat14_SAT_dat.k75-24_1_rule_3.p	1 213 331	4 754 042	2.51
sat14_SAT_dat.k85-24_1_rule_3	2 712 808	5 395 102	2.51
sat14_SAT_dat.k95-24_1_rule_3.p	1 540 071	6 036 162	2.51
sat14_UCG-15-10p1	400 006	1 019 221	2.39
sat14_UCG-15-10p1.p	200 003	1 019 221	2.39
ssmc_ship_001	34 920	34 920	133.0
ssmc_sls	62 729	1 748 122	3.89
ssmc_tmt_unsym2	917 825	917 825	5.0
ssmc_xenon2	157 464	157 464	24.56

Table A.2: Average times when the best solution of the run was found and solution qualities over four runs, with a time limit of 30 minutes each, for all compared algorithms with preprocessing

	ILPPre		OnlineMISPre		CHILS		LSHyMIS		HyMISOnline	
	$\lvert I\rvert$	t	$\lvert I\rvert$	t	$\lvert I\rvert$	t	$\lvert I\rvert$	t	$\lvert I\rvert$	t
ssmc_ABACUS_shell_hd.mtx	2 541	1 800.01	2 541	305.75	**2 542**	**177.25**	2 540	405.87	2 541	485.17
ssmc_BenElechi1.mtx	4 535	1 800.04	4 618	575.31	**4 623**	524.26	4 621	368.61	4 622	**356.55**
ssmc_bips07_1998.mtx	**5 359**	0.07	**5 359**	0.05	**5 359**	0.04	**5 359**	**0.03**	**5 359**	**0.03**
ssmc_ca-CondMat.mtx	**4 057**	0.49	**4 057**	0.45	**4 057**	0.4	**4 057**	0.38	**4 057**	**0.37**
dac2012_superblue2	294 157	1 800.26	294 434	1 763.37	**294 483**	**1 566.5**	294 447	1 756.52	294 445	1 758.5
ssmc_dictionary28.mtx	**29 631**	3.95	**29 631**	0.51	**29 631**	0.58	**29 631**	0.42	**29 631**	**0.41**
ssmc_ex19.mtx	**985**	0.14	**985**	**0.12**	**985**	**0.12**	**985**	**0.12**	**985**	**0.11**
ssmc_fd18.mtx	3 774	1 800.01	4 175	268.98	4 179	**2.8**	**4 182**	709.8	**4 182**	797.17
ssmc_g7jac040sc.mtx	**2 960**	0.33	**2 960**	0.18	**2 960**	0.14	**2 960**	**0.12**	**2 960**	**0.12**
ssmc_garon2.mtx	**400**	1.06	**400**	6.74	**400**	**0.22**	**400**	3.13	**400**	3.04
ISPD98_ibm11	**19 660**	1 800.25	19 658	847.5	19 657	975.12	19 657	994.86	19 656	**503.67**
ISPD98_ibm12	**19 216**	1 800.05	19 212	1 003.01	19 211	**730.15**	19 209	1 072.95	19 210	1 112.25
ssmc_lhr14.mtx	**4 314**	0.28	**4 314**	0.3	**4 314**	0.23	**4 314**	**0.19**	**4 314**	**0.19**
ssmc_lp_pds_20.mtx	**14 743**	1.22	14 743	5.16	**14 743**	**1.0**	**14 743**	1.48	**14 743**	1.49
ssmc_m14b.mtx	8 711	1 800.29	11 637	1 708.34	11 618	1 579.43	**11 669**	1 577.67	11 666	**1 473.06**
ssmc_msc10848.mtx	**88**	2.24	**88**	2.02	**88**	1.93	**88**	1.9	**88**	**1.89**
ssmc_NotreDame_actors.mtx	**20 684**	1 800.01	**20 684**	155.11	20 684	135.94	**20 684**	**48.59**	**20 684**	68.15
ssmc_pdb1HYS.mtx	331	1 800.01	**336**	**81.88**	**336**	344.29	**336**	753.97	**336**	750.44
ssmc_poli3.mtx	**3 523**	0.11	**3 523**	**0.08**	**3 523**	**0.08**	**3 523**	0.09	**3 523**	**0.08**
ssmc_psse2.mtx	**3 098**	0.05	**3 098**	1.11	**3 098**	**0.03**	**3 098**	**0.03**	**3 098**	**0.03**
ssmc_rgg_n_2_18_s0.mtx	19 365	1 800.24	21 684	1 632.4	**21 686**	1 711.19	21 683	1 193.91	21 680	**901.3**
sat14_6s11-opt.cnf	**30 343**	0.95	**30 343**	0.81	**30 343**	2.43	**30 343**	0.44	**30 343**	**0.43**
sat14_6s12.cnf	**31 353**	1.45	31 350	126.25	**31 353**	70.13	31 353	2.4	31 353	**2.28**
sat14_6s130-opt.cnf	**45 782**	1.05	45 779	25.8	**45 782**	**2.32**	45 781	18.41	45 781	40.99
sat14_6s130-opt.cnf.primal	**18 727**	**54.08**	18 720	935.99	18 723	659.25	18 726	382.37	18 726	389.96
sat14_6s133.cnf.primal	**18 577**	**6.86**	18 566	1 121.79	18 568	1 304.05	18 572	1 021.21	18 568	292.72

Table A.3: Average times and solution qualities over four runs, 30 minutes each, for all compared algorithms with preprocessing

	ILPPre		OnlineMISPre		CHILS		LSHyMIS		HyMISOnline	
	$\|I\|$	t	$\|I\|$	t	$\|I\|$	t	$\|I\|$	t	$\|I\|$	t
sat14_6s16.cnf.primal	**11 807**	**4.46**	11 802	314.88	11 805	797.0	11 804	297.41	11 803	217.33
sat14_6s184.cnf	**30 630**	**2.38**	30 624	718.92	30 630	15.5	30 630	821.34	30 630	81.36
sat14_ACG-20-10p1.cnf.dual	176 390	1 800.39	176 292	1 785.45	**176 392**	**1 295.24**	176 384	1 675.82	176 384	1 500.96
sat14_AProVE07-27.cnf.dual	**3 165**	12.2	3 165	410.53	3 165	**9.94**	3 165	905.86	3 165	898.51
sat14_atco_enc1_opt2_05_4.cnf.dual	**7 086**	**190.68**	7 084	1 460.41	7 082	1 470.04	7 085	1 138.8	7 084	1 187.44
sat14_atco_enc1_opt2_10_12.cnf.dual	**4 594**	**197.35**	4 591	715.19	4 588	916.97	4 588	235.4	4 588	233.78
sat14_bob12m09-opt.cnf.dual	**24 209**	**26.97**	24 208	1 288.29	24 209	540.25	24 209	1 304.95	24 208	1 126.29
sat14_dated-10-11-u.cnf	**90 537**	101.38	90 524	8.6	90 527	**2.97**	90 536	360.11	90 534	48.3
sat14_dated-10-17-u.cnf	145 847	1 800.11	146 604	15.54	146 583	**5.78**	**146 627**	746.41	146 621	189.03
sat14_E02F22.cnf.primal	**594**	60.93	594	42.93	594	**35.24**	594	36.82	594	36.66
sat14_manol-pipe-c10nid_i.cnf.primal	**91 911**	**1 008.16**	91 721	1 691.8	91 662	1 751.45	91 780	1 663.32	91 890	983.54
sat14_manol-pipe-c10nidw.cnf.dual	**163 439**	1 800.63	162 426	1 784.62	163 435	**1 612.85**	160 954	1 799.75	161 077	1 799.38
sat14_MD5-30-5.cnf.dual	2 919	1 800.01	2 919	**117.06**	**2 921**	534.86	2 920	208.45	2 920	264.65
sat14_openstacks-p30_3.085-SAT.cnf	**166 047**	68.93	166 023	1 609.79	166 047	**6.53**	166 042	1 044.09	166 041	329.17
sat14_q_query_3_L100_coli.sat.cnf.primal	**131 181**	440.65	131 107	196.98	131 175	1 216.48	131 180	**50.81**	131 180	**12.46**
sat14_q_query_3_L150_coli.sat.cnf	**484 565**	65.75	483 896	22.46	484 557	28.38	484 565	**17.42**	484 565	**17.38**
sat14_SAT_dat.k75-24_1_rule_3.cnf.primal	**474 415**	1 800.37	473 748	1 794.16	474 211	1 775.82	474 247	**1 650.58**	474 226	1 468.18
sat14_SAT_dat.k85-24_1_rule_3.cnf	**1 228 271**	1 801.44	1 224 996	1 792.96	1 227 789	1 758.02	1 227 373	**1 434.08**	1 227 473	1 690.06
sat14_SAT_dat.k95-24_1_rule_3.cnf.primal	**601 989**	1 800.47	600 755	1 794.85	601 715	1 790.87	601 792	**1 624.62**	601 800	1 537.71
sat14_UCG-15-10p1.cnf	132 372	1 800.25	145 458	50.62	145 389	**46.68**	**145 494**	1 407.11	145 494	1 304.34
sat14_UCG-15-10p1.cnf.primal	**39 543**	1 800.18	39 498	1 683.44	39 493	**1 252.49**	39 516	1 443.78	39 521	1 268.1
ssmc_ship_001.mtx	237	1 800.01	**239**	16.33	239	16.76	**239**	**10.89**	**239**	10.95
ssmc_sls.mtx	**54 721**	75.93	54 721	31.82	54 721	27.99	54 721	**27.39**	54 721	**27.33**
ssmc_tmt_unsym.mtx	155 631	1 800.61	176 430	1 797.28	**182 449**	**1 794.66**	182 393	1 795.61	181 774	1 791.28
ssmc_xenon2.mtx	4 258	1 800.08	6 435	901.68	6 427	**485.85**	**6 435**	640.28	**6 435**	724.08

Zusammenfassung

Das Finden maximaler unabhängingiger Mengen ist ein wichtiges und viel erforschten NP-schweres Optimierungsproblem. Eine unabhängige Menge ist ein Teilmenge von Knoten eines Graphen, die paarweise nicht benachbart sind. Die größte unabhängige Menge auf einem Graphen ist die maximale unabhängige Menge (MIS). Es gibt viele Herangehensweisen um das MIS Problem zu lösen. Heuristische Strategien haben sich als effektiv erwiesen, um hochqualitative Lösungen für das Problem auf großen und schwer exakt lösbaren Instanzen zu finden. Obwohl großer Fokus auf dem Finden von großen unabhängigen Mengen liegt, wurde das MIS-Problem auf Hypergraphen, für die jede Kante beliebig viele Knoten enthalten darf, bislang nur wenig untersucht. In dieser Arbeit implementieren wir eine iterative lokale Suche, die auf 2-Verbesserungen basiert, um MIS anzunähern. Genauer untersuchen wir den Einfluss von Reduktionen auf die Leistung des Algorithmus. Eine Reduktion ist in diesem Kontext eine Operation, die einen Graphen gemäß spezifischer Regeln in eine kleinere Instanz transformiert. Vorherige Arbeiten zum Thema zeigen, dass das Reduzieren eines Graphen vor dem Ausführen eines Algorithmus die Leistung von exakten und heuristischen Verfahren verbessern kann. Wir wenden zwischen Schritten der lokalen Suche Reduktionen an, um deren Effizienz zu steigern. Diese Online-Ausführung soll einen Ausgleich zwischen dem positiven Effekt durch Reduktionen und dem zusätzlichen Zeitaufwand, den sie verursachen, schaffen. Für die Online-Reduktionen verwenden wir mehrere, vorher gegebene Reduktionsregeln für Hypergraphen. Wir ergänzen eine ungenaue Reduktion und eine Knotenfaltungsreduktion, die vorher für Graphen vorgeschlagen wurden und übertragen diese auf Hypergraphen. Verglichen mit lokalen Such-Algorithmen auf Graphen finden wir mit unserer hypergraph-spezifischen lokalen Suche meist größere Lösungen in kürzerer Zeit. Die Anwendung von Online-Reduktionen führt zu noch besseren Ergebnissen. Diese sind im Durchschnitt mehrerer Ausführungen für eine einzelne Instanz bis zu 0.2% der besten bekannten Lösung größer. Die durchschnittliche Zeit bis zum Erreichen der besten Lösung eines Algorithmus innerhalb einer Ausführung ist um ca. hundert Sekunden kürzer. Mit unseren Online-Reduktionen reduzieren wir den durchschnittlichen Hypergraphen im Durchschnitt innerhalb von 3.59 Sekunden auf 55% seiner ursprünglichen Größe. Darüber hinaus erweitern wir in dieser Arbeit eine Vorverarbeitung von Hypergraphen, welche in Kombination mit den Reduktionen innerhalb von 7.89 Sekunden den durchschnittlichen Hypergraphen auf 24% einer ursprünglichen Größe reduziert.

Bibliography

[1] Takuya Akiba and Yoichi Iwata. "Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover". In: *Theoretical Computer Science* 609 (2016), pp. 211–225. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2015.09.023. URL: https://www.sciencedirect.com/science/article/pii/S030439751500852X.

[2] Charles J. Alpert. "The ISPD98 Circuit Benchmark Suite". In: *Proceedings of the International Symposium on Physical Design (ISPD)*. New York, NY, USA: Association for Computing Machinery, 1998, pp. 80–85. DOI: 10.1145/274535.274546.

[3] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. "Fast Local Search for the Maximum Independent Set Problem". In: *Journal of Heuristics* 18.4 (2012), pp. 525–547. DOI: 10.1007/s10732-012-9196-4.

[4] Alkida Balliu, Sebastian Brandt, Fabian Kuhn, and Dennis Olivetti. "Distributed Maximal Matching and Maximal Independent Set on Hypergraphs". In: *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2632–2676. DOI: 10.1137/1.9781611977554.ch100. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611977554.ch100. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611977554.ch100.

[5] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. *SAT 2014 Competition*. https://satcompetition.github.io/. 2014.

[6] Endre Boros, Khaled Elbassioni, Vladimir Gurvich, and Leonid Khachiyan. "Generating Maximal Independent Sets for Hypergraphs with Bounded Edge-Intersections". In: *LATIN 2004: Theoretical Informatics*. Ed. by Martín Farach-Colton. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 488–498. DOI: https://doi.org/10.1007/978-3-540-24698-5_52.

[7] Nicolas Bourgeois, Bruno Escoffier, Vangelis T. Paschos, and Johan M. M. van Rooij. "Fast Algorithms for max independent set". In: *Algorithmica* 62.1 (2012), pp. 382–415. ISSN: 1432-0541. DOI: 10.1007/s00453-010-9460-7. URL: https://doi.org/10.1007/s00453-010-9460-7.

[8] Sergiy Butenko, Panos Pardalos, Ivan Sergienko, Vladimir Shylo, and Petro Stetsyuk. “Finding maximum independent sets in graphs arising from coding theory”. In: *Proceedings of the 2002 ACM Symposium on Applied Computing*. SAC ’02. Madrid, Spain: Association for Computing Machinery, 2002, pp. 542–546. ISBN: 1581134452. DOI: 10.1145/508791.508897. URL: https://doi.org/10.1145/508791.508897.

[9] Jakob Dahlum, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. “Accelerating Local Search for the Maximum Independent Set Problem”. In: *15th International Symposium on Experimental Algorithms SEA*. Vol. 9685. Lecture Notes in Computer Science. Springer, 2016, pp. 118–133. URL: https://doi.org/10.1007/978-3-319-38851-9_9.

[10] Mohammad Mehdi Daliri Khomami, Alireza Rezvanian, and Mohammad Reza Meybodi. “Maximum independent set in multiplex social networks and its application to influence maximization”. In: *Scientific Reports* 15 (2025), p. 16322. DOI: 10.1038/s41598-025-99948-z.

[11] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Transactions on Mathematical Software* 38.1 (2011). DOI: 10.1145/2049662.2049663.

[12] Elizabeth D. Dolan and Jorge J. Moré. “Benchmarking optimization software with performance profiles”. In: *Mathematical Programming* 91.2 (2002), pp. 201–213. ISSN: 1436-4646. DOI: 10.1007/s101070100263.

[13] Song Feng, Emily Heath, Brett Jefferson, Cliff Joslyn, Henry Kvinge, Hugh D. Mitchell, Brenda Praggastis, Amie J. Eisfeld, Amy C. Sims, Larissa B. Thackray, Shufang Fan, Kevin B. Walters, Peter J. Halfmann, Danielle Westhoff-Smith, Qing Tan, Vineet D. Menachery, Timothy P. Sheahan, Adam S. Cockrell, Jacob F. Kocher, Kelly G. Stratton, Natalie C. Heller, Lisa M. Bramer, Michael S. Diamond, Ralph S. Baric, Katrina M. Waters, Yoshihiro Kawaoka, Jason E. McDermott, and Emilie Purvine. “Hypergraph models of biological networks to identify genes critical to pathogenic viral response”. In: *BMC Bioinformatics* 22.1 (2021), p. 287. ISSN: 1471-2105. DOI: 10.1186/s12859-021-04197-2. URL: https://doi.org/10.1186/s12859-021-04197-2.

[14] Jieyu Gao, Jie Li, Qihui Wu, Youbiao Wu, and Haikuo Xu. “Distributed Detection of Critical Nodes in Wireless Sensor Networks Using Maximum Independent Set”. In: *2025 IEEE Wireless Communications and Networking Conference (WCNC)*. 2025, pp. 1–5. DOI: 10.1109/WCNC61545.2025.10978779.

[15] Yue Gao, Shuyi Ji, Xiangmin Han, and Qionghai Dai. “Hypergraph Computation”. In: *Engineering* 40 (2024), pp. 188–201. ISSN: 2095-8099. DOI: https://doi.org/10.1016/j.eng.2024.04.017. URL: https://www.sciencedirect.com/science/article/pii/S2095809924002510.

[16] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. "Scalable High-Quality Hypergraph Partitioning". In: *ACM Transactions on Algorithms* 20.1 (2024), 9:1–9:54. DOI: 10.1145/3626527.

[17] Ernestine Großmann, Sebastian Lamm, Christian Schulz, and Darren Strash. "Finding Near-Optimal Weight Independent Sets at Scale". In: *Journal of Graph Algorithms and Applications* 28.1 (Nov. 2024), pp. 439–473. DOI: 10.7155/jgaa.v28i1.2997. URL: https://jgaa.info/index.php/jgaa/article/view/2997.

[18] Ernestine Großmann, Kenneth Langedal, and Christian Schulz. "Accelerating Reductions Using Graph Neural Networks and a New Concurrent Local Search for the Maximum Weight Independent Set Problem". In: *arXiv preprint arXiv:2412.14198* (2024).

[19] Ernestine Großmann, Christian Schulz, Darren Strash, and Antonie Wagner. *Data Reductions for the Strong Maximum Independent Set Problem in Hypergraphs*. 2026. arXiv: 2602.10781 [cs.DS]. URL: https://arxiv.org/abs/2602.10781.

[20] Andrea Grosso, Marco Locatelli, and Wayne Pullan. "Simple ingredients leading to very efficient heuristics for the maximum clique problem". In: *Journal of Heuristics* 14.6 (2008), pp. 587–612. ISSN: 1572-9397. DOI: 10.1007/s10732-007-9055-x.

[21] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2026. URL: https://www.gurobi.com.

[22] Pierre Hansen, Nenad Mladenović, and Dragan Urošević. "Variable neighborhood search for the maximum clique". In: *Discrete Applied Mathematics* 145.1 (2004). Graph Optimization IV, pp. 117–125. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2003.09.012. URL: https://www.sciencedirect.com/science/article/pii/S0166218X04000708.

[23] Thomas Hofmeister and Hanno Lefmann. "Approximating maximum independent sets in uniform hypergraphs". In: *Mathematical Foundations of Computer Science 1998*. Ed. by L. Brim, J. Gruska, and J. Zlatuška. Vol. 1450. Lecture Notes in Computer Science. Springer, 1998, pp. 293–302. DOI: 10.1007/BFb0055806.

[24] Deborah Joseph, Joao Meidanis, and Prasoon Tiwari. "Determining DNA sequence similarity using maximum independent set algorithms for interval graphs". In: *Algorithm Theory — SWAT '92*. Ed. by Otto Nurmi and Esko Ukkonen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 326–337. ISBN: 978-3-540-47275-9. DOI: https://doi.org/10.1007/3-540-55706-7_29.

[25] George Karypis and Vipin Kumar. "A Hypergraph Partitioning Package". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 1.1 (1998).

[26] Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Renato F. Werneck. "Finding near-optimal independent sets at scale". In: *J. Heuristics* 23.4 (2017), pp. 207–229. DOI: `10.1007/s10732-017-9337-x`.

[27] Elena Losievskaja. "Approximation algorithms for independent set problems on hypergraphs". PhD thesis. PhD thesis. Reykjavík, Iceland: Reykjavík University, 2011. URL: `https://opinvisindi.is/items/b7f3c915-5c17-4025-8879-2ace7b64ccfd`.

[28] Patric R.J. Östergård. "A fast algorithm for the maximum clique problem". In: *Discrete Applied Mathematics* 120.1 (2002). Special Issue devoted to the 6th Twente Workshop on Graphs and Combinatorial Optimization, pp. 197–207. ISSN: 0166-218X. DOI: `https://doi.org/10.1016/S0166-218X(01)00290-6`. URL: `https://www.sciencedirect.com/science/article/pii/S0166218X01002906`.

[29] J.M Robson. "Algorithms for maximum independent sets". In: *Journal of Algorithms* 7.3 (1986), pp. 425–440. ISSN: 0196-6774. DOI: `https://doi.org/10.1016/0196-6774(86)90032-5`. URL: `https://www.sciencedirect.com/science/article/pii/0196677486900325`.

[30] Robert Endre Tarjan and Anthony E. Trojanowski. "Finding a Maximum Independent Set". In: *SIAM Journal on Computing* 6.3 (1977), pp. 537–546. DOI: `10.1137/0206038`. URL: `https://doi.org/10.1137/0206038`.

[31] Jacques Verstraete and Chase Wilson. "Independent Sets in Hypergraphs". In: *Random Structures & Algorithms* 68.1 (2026), e70047. DOI: `https://doi.org/10.1002/rsa.70047`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.70047`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/rsa.70047`.

[32] Natarajan Viswanathan, Charles Alpert, Cliff Sze, Zhuo Li, and Yaoguang Wei. "The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite". In: *Proceedings of the Annual Design Automation Conference (DAC)*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 774–782. DOI: `10.1145/2228360.2228500`.

[33] Mingyu Xiao and Hiroshi Nagamochi. "Exact algorithms for maximum independent set". In: *Information and Computation* 255 (2017), pp. 126–146. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/j.ic.2017.06.001`. URL: `https://www.sciencedirect.com/science/article/pii/S0890540117300950`.