Karlsruhe Institute of Technology

# Engineering Generalized Reductions for the Maximum Weight Independent Set Problem

Master's Thesis of


Alexander Gellner

at the Department of Informatics
Institute of Theoretical Computer Science


Reviewer: Prof. Dr. Peter Sanders
Second reviewer: Prof. Dr. Dorothea Wagner
Advisors: M.Sc. Sebastian Lamm
Priv.-Doz. Dr. Christian Schulz, University of Vienna
Dr. Darren Strash, Hamilton College
Dr. Bogdán Zaválnij, Alfréd Rényi Institute of Mathematics

January 30, 2019 – June 30, 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

**Karlsruhe, June 30, 2020**

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(Alexander Gellner)

# Abstract

The maximum weight independent set problem asks on a vertex-weighted, undirected graph for a subset of pairwise non-adjacent vertices whose total weight is maximum. This problem has applications in many areas such as map labeling [9, 22], coding theory [10, 44] or combinatorial auctions [57].

Many of these real-world instances are very large and consist of thousands up to millions of vertices, making them infeasible for many exact algorithms [33]. A very well-established approach in practice is kernelization or data reduction, where a problem instance is transformed into a smaller equivalent instance [3, 33, 52]. This process is performed by iterative application of reduction rules until an irreducible graph is created. The so-called branch-and-reduce paradigm is an exact solution method, which has proven itself in the past and is nowadays used in state-of-the-art solvers. Thereby the input instance is first reduced in size by data reduction, followed by branching into one or more subproblems by case distinctions and further application of reduction rules [3, 33]. In the past, a large number of reduction rules have been developed for the maximum weight independent set problem, which allows a large number of real-world instances to be solved. However, on some instances that are for example derived from map labeling problems, these rules still fail to produce small irreducible graphs and hence compute reduced graphs that are still hard to solve [33].

To tackle this problem, in this thesis we develop new data reduction techniques that are able to reduce previously irreducible graphs by use of generalized reduction rules. In this regard, we investigate the theoretical and practical potential of the weighted stability number reduction (struction) [19, 61]. In contrast to traditional reduction rules, the struction can not only decrease but also increase the size of the graph. Besides an algorithm that only uses special cases of the struction which do not increase the graph size, we develop an additional algorithm that exploits the full potential of the struction by also allowing struction applications that blow-up the graph. Interestingly, while these reductions temporarily increase the graph size, they also open up the space for further reduction rule application and thus are able to obtain smaller irreducible graphs after all.

Our experiments on real-world instances show, that compared to the best state-of-the-art reduction algorithm, on some instance families we are able to completely reduce previously irreducible graphs with several thousand vertices to an empty graph or to obtain reduced graphs which are up to two orders of magnitude smaller. Moreover we are able to solve 12.5% more instances optimally than the best branch-and-reduce algorithm and are on average a factor five faster on instances that the previous state-of-the-art solver could solve. Compared to other state-of-the-art local searches, on many instance families we are able to either find better solutions or equivalent solutions in less time.

# Zusammenfassung

Das Problem der gewichtsmaximalen unabhängigen Menge sucht in einem knotengewichteten, ungerichteten Graphen nach einer Teilmenge paarweise nicht benachbarten Knoten mit maximalem Gesamtgewicht und findet in Bereichen wie Kartenbeschriftung [9, 22], Codierungstheorie [10, 44] oder kombinatorischen Auktionen [57] Anwendung.

Viele dieser Instanzen sind sehr groß und bestehen aus tausenden bis zu millionen von Knoten, wodurch sie für viele exakte Algorithmen [33] nicht lösbar sind. Ein in der Praxis bewährter Ansatz ist die Kernfindung oder Datenreduktion, bei der eine Probleminstanz in eine kleinere, äquivalente Instanz transformiert wird [3, 33, 52]. Dieser Prozess wird durch iterative Anwendung von Reduktionsregeln durchgeführt, bis ein irreduzibler Graph entsteht. Das so genannte branch-and-reduce Paradigma ist eine exakte Lösungsmethode, die sich in der Vergangenheit bewährt hat und in aktuellen Lösungsverfahren eingesetzt wird. Dabei wird die Eingabeinstanz zunächst durch Datenreduktion verkleinert, gefolgt von einer Aufspaltung in ein oder mehrere Teilprobleme durch Fallunterscheidungen und weitere Anwendung von Reduktionsregeln [3, 33]. In der Vergangenheit wurde eine große Anzahl von Reduktionsregeln für das Problem der gewichtsmaximalen unabhängigen Menge entwickelt, wodurch eine große Anzahl von Instanzen aus der Praxis gelöst werden kann. Bei einigen Instanzen, die beispielsweise von Kartenbeschriftungsproblemen stammen, scheitern diese Regeln jedoch noch immer daran, kleine irreduzible Graphen zu erzeugen und berechnen reduzierte Graphen, die weiterhin schwer zu lösen sind [33].

Zur Bewältigung dieses Problems entwickeln wir in dieser Arbeit neue Datenreduktionstechniken, die bisher irreduzible Graphen durch verallgemeinertere Reduktionsregeln reduzieren können. Dabei untersuchen wir das theoretische und praktische Potenzial der sogenannten Struction [19, 61]. Im Gegensatz zu traditionellen Reduktionsregeln kann diese den Graphen nicht nur verkleinern, sondern auch vergrößern.

Neben einem Algorithmus, der nur Structions verwendet, welche die Graphgröße nicht erhöhen, entwickeln wir zusätzlich einen Algorithmus, der das volle Potential der Struction ausschöpft, indem er auch Anwendungen der Struction erlaubt, die den Graphen aufblähen. Interessanterweise erhöhen diese Reduktionen zwar vorübergehend die Graphengröße, öffnen gleichzeitig aber auch den Raum für die Anwendung weiterer Reduktionsregeln, wodurch sich letztendlich kleinere irreduzible Graphen erhalten lassen.

Unsere Experimente zeigen, dass wir im Vergleich zum aktuell besten Reduktionsalgorithmus zuvor irreduzible Graphen mit mehreren tausend Knoten oftmals vollständig reduzieren können oder reduzierte Graphen erhalten, welche um bis zu zwei Größenordnungen kleiner sind. Darüber hinaus sind wir in der Lage, 12,5% mehr Instanzen als der beste branch-and-reduce Algorithmus optimal zu lösen und zuvor lösbare Instanzen im geometrischen Mittel um einen Faktor fünf schneller zu lösen. Im Vergleich zu modernen lokalen Suchen sind wir bei vielen Instanzfamilien in der Lage, entweder bessere Lösungen oder gleichwertige Lösungen in kürzerer Zeit zu finden.

# Contents


**Abstract** i

**Zusammenfassung** iii

**1. Introduction** 1
- 1.1. Motivation 1
- 1.2. Contribution 2
- 1.3. Thesis Structure 2

**2. Preliminaries** 3
- 2.1. Basic Definitions 3
- 2.2. Maximum Weight Independent Set And Related Problems 3
- 2.3. Reduction Rules and Kernelization 4

**3. Related Work** 5
- 3.1. Exact Methods 5
- 3.2. Heuristic Methods 6
- 3.3. Struction 8

**4. Branch-And-Reduce Framework** 11
- 4.1. Basic Reduction Algorithm 11
  - 4.1.1. Reduction Rules 11
  - 4.1.2. Reduction Rule Order 14

**5. Struction** 17
- 5.1. Unweighted Struction 17
- 5.2. Weighted Struction Variants 18
  - 5.2.1. Original Weighted Struction 18
  - 5.2.2. Modified Weighted Struction 21
  - 5.2.3. Extended Weighted Struction 22
  - 5.2.4. Extended Reduced Weighted Struction 24
- 5.3. Relationship To Other Reduction Rules 27
  - 5.3.1. Clique Neighborhood Removal ⊆ Extended Weighted Struction$_{C=C'}$ 27
  - 5.3.2. Generalized Fold ⊆ Extended Weighted Struction$_{C=C'}$ 28
  - 5.3.3. Isolated Weight Transfer ⊆ Extended Weighted Struction$_{C=C'}$ 28
  - 5.3.4. Weighted Isolated Vertex Removal ⊆ Isolated Weight Transfer 29
  - 5.3.5. Degree Two Fold ⊆ Generalized Fold 29
  - 5.3.6. Neighborhood Removal ⊆ Clique Neighborhood Removal 29
  - 5.3.7. New Reducible Graph Structures 30

**6. Efficient Data Reduction Via Struction** 31
6.1. Non-Increasing Reduction Algorithm . . . . . 32
6.2. Cyclic Blow-Up Algorithm . . . . . 33
6.2.1. Blow-Up Phase . . . . . 34
6.2.2. Accept Strategies . . . . . 37
6.2.3. Cycle Avoidance Strategies . . . . . 37
6.2.4. Stopping Criteria . . . . . 37

**7. Evaluation** 39
7.1. Experimental Setup . . . . . 39
7.1.1. Environment . . . . . 39
7.1.2. Datasets . . . . . 39
7.1.3. Methodology . . . . . 40
7.1.4. Experimental Design . . . . . 41
7.2. Parameter Tuning . . . . . 42
7.2.1. Non-Increasing Reduction Algorithm . . . . . 42
7.2.2. Cyclic Blow-Up Algorithm . . . . . 45
7.3. Comparison With Existing Algorithms . . . . . 50
7.3.1. Comparison With Branch-And-Reduce Framework . . . . . 51
7.3.2. Comparison With State-Of-The-Art Algorithms . . . . . 56

**8. Discussion** 63
8.1. Conclusion . . . . . 63
8.2. Future Work . . . . . 64

**A. Appendix** 65
A.1. Basic Graph Property Tables . . . . . 65
A.2. Reduced Graph Size Convergence Plots . . . . . 68
A.3. Time To Solve And Reduced Graph Size Tables . . . . . 71
A.4. Best Solution Tables . . . . . 75
A.5. Solution Quality Convergence Plots . . . . . 79

**Bibliography** 83

# 1. Introduction

## 1.1. Motivation

The *maximum weight independent set* problem (MWIS) is an $\mathcal{NP}$-hard problem [21] which has many practical applications in areas such as map labeling [9, 22], coding theory [10, 44] or combinatorial auctions [57]. For a weighted graph, it searches for an independent set, i.e. pairwise non-adjacent vertices, whose total weight is maximum.

In algorithmic cartography, weighted independent sets can be used to automatically generate high-quality map labelings [9, 22]: A set of potential label candidates is given, each with an importance $i$, which e.g. for city name labels is chosen proportional to the number of inhabitants. The goal is to select a set of non-overlapping labels and thereby maximize the sum of the label weights. This problem can be solved by finding an MWIS on the so-called *label conflict graph*, having a vertex with weight $i$ for each label and an edge for each pair of overlapping labels.

In coding theory [10, 44], large codes with Hamming distance of $d$ can be created by first partitioning the space of all code-words into disjoint subsets, so called orbits. For each orbit, a vertex is created with a weight corresponding to the number of code-words it contains, ignoring orbits with code-words that are less than $d$ distant from each other. Edges are inserted between orbits if the first orbit contains a code-word with smaller distance than $d$ to a code-word of the second orbit. The number of codewords is thus equivalent to the weight of an independent set in this graph.

Other real-world problems can be found in computational biology in the alignment of biological networks [6], workload scheduling for energy-efficient scheduling of disks [15], computer vision [40], wireless communication [59] and protein structure prediction [41].

Many of these real-world instances are very large, consisting of several thousand up to millions of vertices, making them intractable for optimal solution methods [33]. A preprocessing method used in practice is *data reduction* or *kernelization* [16], which reduces the problem to a smaller equivalent instance. By iterative application of *reduction rules*, the input graph is reduced in size until an irreducible graph is obtained. This irreducible graph is called a *kernel* if it has size bounded by a function of a parameter. Then a solution is calculated on this irreducible graph and extended to a solution of the original instance by undoing the reduction rules.

For the maximum weight independent set there are several reduction rules which are able to calculate small irreducible graphs on a large number of instances, but still fail on some of them [33]. By researching further and more general reduction rules, we hope to be able to calculate smaller equivalent instances for these graphs as well and thus make them feasible.

## 1.2. Contribution

In this thesis we give a new contribution to data reduction techniques for the maximum weight independent set problem. We investigate both the theoretical and practical potential of an already known reduction rule, the stability number reduction (struction). Several variants of this already exist, but none of them have been tested extensively on real-world instances so far.

In a theoretical analysis we compare these variants with already existing reduction rules proposed by Lamm et al. [33]. We find that two of these variants subsume six of the eight rules on their own and can also reduce many new, previously irreducible graphs.

We then design two new reduction algorithms that use the struction and other reduction rules of the framework. The first is a simple extension that uses a restricted variant of the struction as a new rule that always reduces the graph size or keeps it the same. In the second algorithm, we further exploit the potential of the struction by also allowing the application of structions that may increase the number of vertices. The motivation behind this is to expand the reduction space and create graphs that can be reduced again. Since common approaches in practice usually try to reduce via incremental application of reduction rules, which always result in a smaller graph, we explore a far more general technique with this approach.

In a subsequent evaluation, we show that both algorithms can calculate significantly smaller equivalent instances on many real-world graphs. Finally, we find that we are now able to optimally solve some real-world instances, which were previously infeasible, using the branch-and-reduce framework by Lamm et al. [33] combined with our reductions.

## 1.3. Thesis Structure

We start this thesis with a brief overview of the fundamentals and basic notations in Chapter 2. Then we introduce related work on weighted independent sets and different struction variants in Chapter 3, which we want to explore in this work. Chapter 4 gives us an overview of the branch-and-reduce framework of Lamm et al. [33], providing the foundation for our study. After that, Chapter 5 introduces the different variants of weighted struction and shows relationships to other existing reduction rules. Two new struction based reductions are introduced in Chapter 6. In the following evaluation in Chapter 7 these algorithms are tested on various real-world instances. Finally, we conclude the work with a short summary and an outlook on future work in Chapter 8.1.

# 2. Preliminaries

In following chapter we introduce some notations that are used in this work. We also give a brief overview of the maximum independent set problem and related problems. Finally, a brief outline of the kernelization and reduction rules is given.

## 2.1. Basic Definitions

A graph $G = (V, E)$ consists of a *vertex set* $V$ and an *edge set* $E \subset V \times V$. It is called *undirected* if for each edge $(u, v) \in E$ the edge set also set contains the corresponding edge $(v, u)$, i.e. $(u, v) \in E \Leftrightarrow (v, u) \in E$. In this thesis we only consider undirected graphs without self loops, i.e. $(v, v) \notin E$, and therefore we describe edges by sets $\{u, v\}$.

In addition, we denote a graph as (vertex-)*weighted* if a positive scalar weight $w(v)$ is assigned to each vertex $v \in V$, i.e. we have a function $w : V \rightarrow \mathbb{R}^+$. The weight of a vertex set $X \subset V$ describes the sum of all its vertices' weights and is denoted by $w(X) = \sum_{v \in X} w(x)$.

A graph $G' = (V', E')$ is called *subgraph* of $G = (V, E)$ if $V' \subset V$ and $E' \subset E \cap (V' \times V')$ holds. Given a vertex set $U \subset V$ the *induced subgraph* of $G$ is the graph $G' = (U, E')$ with $E' = E \cap (U \times U)$ which is denoted by $G[U]$.

Two vertices $u, v$ are called *adjacent* if there is an edge between them, i.e. $\{u, v\} \in E$. As the *neighborhood* $N(v)$ of a vertex $v$ we describe the set of all vertices adjacent to $v$. By $N[v] = N(v) \cup \{v\}$ the *closed neighborhood* and by $\overline{N}(v) = V \setminus N[v]$ the *non-neighborhood* of $v$ are denoted. We denote the *degree* of a vertex $v$, the number of its neighbors, by $\delta(v) = |N(v)|$. Furthermore, we define the neighborhood of a vertex set $U \subseteq V$ as $N(U) = \bigcup_{v \in U} N(v) \setminus U$ and the closed neighborhood as $N[U] = N(U) \cup U$.

## 2.2. Maximum Weight Independent Set And Related Problems

For a given graph $G = (V, E)$ a vertex set $I \subset V$ is an independent set if all vertices $v \in I$ are pairwise non adjacent, that is $\forall\, u, v \in I : \{u, v\} \notin E \vee u = v$. An independent set is called *maximal* if it is not a subset of another independent set and *maximum* if no other independent set of greater cardinality exists. The independence number $\alpha(G) = |I|$, sometimes also called stability number, of a graph $G$ is the cardinality of a maximum independent set $I$ in $G$.

For a weighted graph $G$ an independent set $I$ has *maximum weight*, if there is no independent set $I'$ in $G$ with a weight $w(I')$ greater than $w(I)$. The weighted independence number $\alpha_w(G) = w(I)$ of a weighted graph $G$ is defined as the weight of a maximum weight independent set $I$ in $G$. For a given weighted graph $G$, the *maximum weight independent set problem* (MWIS) seeks a maximum weight independent set $I$ in $G$.

Closely related problems to the maximum weight independent set problem are the maximum weight clique and minimum weight vertex cover problems. A vertex set $C \subset V$ forms a clique in a graph $G = (V, E)$ if all vertices $v \in C$ are pairwise adjacent, i.e. $\forall\, u, v \in C : \{u, v\} \in E$ applies. In addition, a vertex set $V_C \subset V$ is a vertex cover if each edge is covered by at least one vertex $v \in V_C$, thus $\forall \{u, v\} \in E : u \in V_C \vee v \in V_C$ holds. Analogous to the maximum weight independent set problem, the *minimum weight vertex cover* (MWVC) looks for a vertex cover of minimum weight and the *maximum weighted clique* problem (MWC) for a clique of maximum weight. For any weighted graph $G = (V, E)$ a maximum weight independent set $I$ is a maximum weight clique in the complement graph $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{\{u, v\} \subset V : u \neq v \wedge \{u, v\} \notin E\}$ and $V \setminus I$ is a minimum weight vertex cover in $G$ [43, 58].

## 2.3. Reduction Rules and Kernelization

A frequently used approach for solving maximum (weight) independent set instances (and analogously also minimum (weight) vertex cover instances) is called *kernelization*, which was originally introduced to reduce the complexity of exact solution methods [3, 30, 31, 33, 52], but is nowadays also used in heuristic methods [14, 17, 36]. Besides the independent set problem, kernelization is also used in many other areas, such as the dominating set problem [1], minimum cut problem [27, 28, 29], maximum cut problem [20] or multiterminal cut problem [26].

The goal of kernelization is to reduce the complexity of a given instance in polynomial time by solving "simple parts" of the graph, leaving the hard to solve core of the problem, called the *kernel*. This is achieved by using reduction rules, which e.g. add vertices to the independent set if they are proven to belong to a maximum (weight) independent set. How hard a given instance is actually to solve is generally difficult to say. Motivated by the fact that exact solution methods have an asymptotic running time exponential in the number of vertices, the goal is usually to reduce the number of vertices of an instance [3, 33, 52].

In the following a formal definition for reduction rules and kernelization is given [16]. We generally consider parameterized problems that are defined as languages $L \subseteq \Sigma^* \times \mathbb{N}$ over an input alphabet $\Sigma$. An instance $(I, k) \in L$ is a yes-instance if it is contained in a set $Q \subseteq L$, i.e. $(I, k) \in Q$. For the maximum weight independent set problem, we obtain a parameterized problem by considering the corresponding decision problem. For a weighted graph, this problem asks whether there is an independent set with weight greater than or equal to $k$. An instance is therefore a yes-instance if such a set exists.

For a given parameterized problem, a *reduction rule* is a function $\phi : \Sigma^* \times \mathbb{N} \rightarrow \Sigma^* \times \mathbb{N}$, which converts a problem instance $(I, k)$ into an equivalent problem instance $(I', k')$ in polynomial time in $|I|$ and $k$. Two problem instances $(I, k), (I', k')$ are called *equivalent* if $(I, k) \in Q \Leftrightarrow (I', k') \in Q$ holds.

On this foundation, a preprocessing algorithm is often used to transform an instance $(I, k)$ into an equivalent instance $(I', k')$ in polynomial time by iterative application of reduction rules. Such an algorithm is called a *kernelization algorithm*, if for any instance $(I, k)$ the size of the output instance $(I', k')$ is bounded by a function $g(k)$, i.e. $|I'| + k' \leq g(k)$. Kernelization algorithms are called *polynomial (linear)* if $g(k)$ is polynomial (linear) in $k$.

# 3. Related Work

Motivated by the $\mathcal{NP}$-completeness of the maximum independent set and maximum weighted independent set problem [21], existing work distinguishes between two major algorithm classes, namely exact and heuristic methods.

While exact methods always compute optimal solutions in exponential worst case running time and prove that there is no better solution, heuristic methods usually run in polynomial time, but generally have no guarantee for the quality of the computed solutions. In the following we will first give a short overview of existing work on both exact and heuristic procedures, especially outlining how kernelization and preprocessing methods are used in state-of-the-art algorithms. Furthermore, we examine the mainly theoretically oriented work on the struction, which is a reduction rule that forms the basis of this this work.

## 3.1. Exact Methods

Exact algorithms usually compute their optimal solutions by systematically exploring the solution space. So-called *branch-and-bound* methods [46, 54] achieve this by case distinctions in which vertices are either included into the current solution or excluded from it, branching into two or more subproblems and resulting in a search tree. In past work, branch-and bound methods have been improved by new branching schemes or better pruning methods using upper and lower bounds to exclude certain subtrees. In practice, exact state-of-the-art algorithms are thus able to calculate optimal solutions on instances with several hundred to a few thousand vertices [7, 8, 35]. In the following we want to have a look on important works that follow the branch-and-bound paradigm.

Balas and Yu [8] presented an algorithm for finding maximum cliques. This algorithm introduced a very generic branching scheme, which is used in many algorithms for the weighted case with small adjustments. For this purpose, they look for certain vertex sets in the graph and for each vertex they branch into a new subproblem.

Babel [7] subsequently presented an algorithm that calculates upper and lower bounds using a weighted clique heuristic. They are also able to derive a branching scheme from a weighted clique calculated by their heuristic as a special kind of the scheme by Balas and Yu.

Warren and Hicks [54] also presented three more branch-and-bound algorithms, all using weighted clique covers and the branching scheme introduced by Balas and Yu. The first extends the Babel algorithm by accelerating it using a more sophisticated data representation. The second is an adaptation of the algorithm of Balas and Yu, which uses a weighted clique heuristic that yields structurally similar results the heuristic of Babel. The

last algorithm is a hybrid version that combines both algorithms and is able to compute solutions on graphs with hundreds of vertices.

An important part of exact methods are reduction rules which are used in the so-called *branch-and-reduce* paradigm [3]. It improves the worst-case runtime of branch-and-bound algorithms by application of reduction rules to the current graph before each branching step. For the unweighted case, a large number of branch-and-reduce algorithms have been developed in the past. However, for a long time, virtually no weighted reduction rules were known, which is why hardly any branch-and-reduce algorithms exist for the maximum weighted independent set problem.

To the best of our knowledge, the first and only branch-and-reduce algorithm for the weighted case was recently presented by Lamm et al. [33] The authors first introduce two meta-reductions called neighborhood removal and neighborhood folding, from which they derive a new set of weighted reduction rules. On this foundation a branch-and-reduce algorithm is developed using weighted clique covers similar to the approach in [54] for upper bounds and an adapted version of the ARW local search [5] for lower bounds. The experimental evaluation shows that their algorithm can solve the majority of the tested real-world instances and outperforms heuristic algorithms on a large number of instances.

Furthermore, algorithms can be found which follow the *branch-and-cut* [49], *branch-and-price* [56] or *branch-price-and-cut* [55] paradigm. These are also extensions of branch-and-bound paradigm, however, these will not be discussed in the following.

There are some other exact procedures, which are based on the reformulation into other $\mathcal{NP}$-complete problems, for which a variety of solvers already exist. For instance, Xu et al. [58] recently developed an algorithm called SBMS, which calculates an optimal solution for a given MWVC instance by solving a series of SAT instances.

## 3.2. Heuristic Methods

A widely used approach for heuristic methods is local search, which usually first computes an initial solution and then tries to improve it by simple insertion, removal or swap operations. Although in theory local searches generally offer no guarantees for the solutions quality, in practice they often find high quality solutions significantly faster than exact procedures.

For unweighted graphs, the iterated local search by Andrade et al. [5], often referred to as ARW in the literature, is a very successful state-of-the-art heuristic. It is based on so-called $(1, 2)$-swaps which remove one vertex from the current solution and add two new vertices to it, thus improving the current solution by one. Their algorithm uses special data structures which find such a $(1, 2)$-swap in linear time in the number of edges or prove that none exists. A performance optimization presented here consists of an incremental update strategy, where only vertices within a candidate list have to be considered for removal during a swap. Consequently, a vertex only needs to be checked again after a certain change has occurred in its neighborhood. To prevent the search from getting stuck in a local minimum, ARW also contains a perturbation operation that forces vertices into the current solution, removing neighboring vertices from the solution.

The hybrid iterated local search (HILS) by Nogueira et al. [43] adapts the ARW for weighted graphs. In addition to weighted $(1, 2)$ swaps, it also uses $(\omega, 1)$ swaps that include one vertex $v$ into the current solution and exclude $\omega$ vertices from it, namely all neighbors of $v$ contained in the solution. Such swaps are only allowed as long as they improve the solution, i.e. the the sum of the excluded vertices' weights is smaller than the weight of $v$. These two types of neighborhoods are explored separately using variable neighborhood descent (VND), i.e. switching to the next neighborhood as soon as the current one fails to improve the solution and otherwise switching back to the first neighborhood. In practice, the algorithm finds all known optimal solutions on well-known benchmark instances within milliseconds and thus outperforms other state-of-the-art local searches.

Two other local searches for the equivalent minimum weighted vertex cover problem are presented by Cai et al. [12], which extend the existing FastWVC heuristic [37] by dynamic selection strategies for vertices to be removed from the current vertex cover. While previous algorithms always select vertices based on a single scoring function, the first approach DynWVC1 dynamically switches between two different scoring functions. DynWVC2 extends this algorithm by dynamically determining the number of removed vertices within an iteration. In practice, DynWVC1 outperforms previous MWVC heuristics on map labeling instances and large scale networks, and DynWVC2 provides further improvements on large scale networks but performs worse on map labeling instances than DynWVC1.

Over the last few years, reduction rules have been combined with local searches. For the unweighted case, Dahlum et al. [17] accelerated the ARW algorithm with their OnlineMIS approach, where simple reduction rules are applied on-the-fly during the local search. In addition, they introduce another approach, KerMIS, which applies a set of reduction rules, then removes high degree vertices and performs ARW on the resulting graph.

Based on this approach, Chang et al. [14] developed two algorithms with linear and near-linear time complexity. For the LinearTime algorithm, the authors develop new linear-time reduction rules that are special cases of the degree-two-fold rule [3] while for NearLinearTime they additionally present an incremental version of the dominance rule [3]. Applying these rules, both algorithms compute an initial irreducible graph and then calculate a solution for it in an iterative fashion by removing high degree vertices until reduction rules can be applied again.

Most recently, Li et al. [36] presented a local search for the minimum weight vertex cover problem called NuMWVC, which applies reduction rules during the construction phase of the initial solution. Furthermore, they adapt the configuration checking approach [13] to the MWVC problem which is used to reduce cycling, i.e. returning to a solution that has been visited recently. Finally, they develop a technique called self-adaptive-vertex-removing, which dynamically adjusts the number of removed vertices per iteration, similar to DynWVC2. Experiments show that NuMWVC outperforms state-of-the-art algorithms on both massive graphs and real-world problems, although a comparison between DynWVC and NuMWVC is still pending.

## 3.3. Struction

In general, the struction method can be classified as a reduction rule that reduces the independence number of a graph. In the literature further similar transformations like the conic reduction [39] or clique reduction [38] can be found, however in the following the focus will be on the struction and related variations.

Originally the struction (STability number RedUCTION) was introduced by Ebenegger et al. [19] and was later improved by Alexe et al. [4]. In fact, this method is a graph transformation for unweighted graphs, which reduces their stability number by exactly one. Therefore an arbitrary vertex and its neighborhood is removed from the graph and new vertices are inserted for each non-adjacent pair of neighbors, which are connected to the rest of the graph by certain edges. In the following work, we refer to this struction variant as original struction (see Section 5.1). By successive application of the struction, the independence number of a graph can be determined. However, since the number of vertices in the transformed graph can increase in comparison to the original graph, this method generally has exponential memory requirements and thus exponential runtime. In their evaluation, the authors find that by application of further reduction rules before each struction execution this vertex increase can be slowed down in some cases, or can even result in a decrease occasionally. Ebenegger et al. also show that there is an equivalence between finding a maximum weight independent set and maximizing a pseudo Boolean function, i.e. a real-valued function with Boolean variables, which allows to derive the struction as a special case. Finally, the authors present a generalization of the struction to weighted graphs (original weighted struction, see Section 5.2.1). This variant creates the same new vertices as in the unweighted case, but is only applicable to vertices with minimal weight with respect to their neighbors. Furthermore, this variant only removes the vertex to which the struction is applied and retains its neighborhood.

On this basis some theoretical algorithms with polynomial time complexity for special graph classes have been developed [24, 25], using further reduction rules and a careful selection of vertices on which the struction is applied.

Hoke and Troyon [32] developed another form of the weighted struction, using the same equivalence found by Ebenegger et al. [19]. They take advantage of the fact that the struction can be interpreted as a special case of the `Basic Algorithm`, which is a general method for finding a maximum of a pseudo-boolean function. Thus they are able to derive the *revised weighted struction*, which, however, is only applicable in claw-free graphs, i.e. graphs that do not contain a three-leaf star graph. This transformation also removes a vertex $v$ and its neighborhood, but is able to create fewer new vertices, since these are only created for pairs of non-adjacent neighbors whose combined weight is greater than the weight of $v$.

Most recently, Zavalnij [61] introduced three more variants of the weighted struction. The first version (modified weighted struction, see Section 5.2.2) deals with the fact that in the original weighted struction, an MWIS in the transformed graph consists of more vertices than in the original graph. By different weight assignments for the new vertices and inserting additional edges, this variant ensures that these two numbers are the same. The second version (extended weighted struction, see Section 5.2.3) is a generalization of the revised weighted struction, as it can be applied to general graphs on any vertex $v$

without the need to fulfill certain weight constraints. This variant creates a new vertex for each independent set in the neighborhood of $v$ whose weight is greater than $v$. By creating new vertices for only a specific subset of these sets, the last version (extended reduced weighted struction, see Section 5.2.3) reduces the number of new vertices.

Up to now, only little effort has been invested in a practical evaluation of the different struction variants: Ebenegger et al. and Alexe et al. evaluated the struction only on small graphs with less than a hundred vertices for the unweighted case [4, 19]. Furthermore, for the weighted case none of the presented struction variants has been evaluated so far [4, 19, 32, 61]. Therefore, a detailed evaluation (on real world instances) especially for the weighted case is still pending.

# 4. Branch-And-Reduce Framework

In this chapter a short overview of the branch-and-reduce framework by Lamm et al. [33] is given, which forms the basis for further reduction algorithms in the following chapter. For this purpose, its individual components will be explained and the in particular the used reductions will be discussed.

An overview of the framework is given in Algorithm 1 (cf. [33], Algorithm 1), which outputs the weight of a maximum weight independent set for a given graph. In practice, the framework computes the actual MWIS with slight adaptation of this algorithm. During the execution the weight of the best independent set found so far and the current solution weight are maintained. Following the branch-and-reduce paradigm, we obtain an irreducible graph from our current graph by application of reduction rules. In the following work we refer to the used reductions as *basic reduction algorithm*, which we will discuss in more detail in Section 4.1. At the beginning of the algorithm we first obtain an irreducible graph on which we perform a local search to get a high quality initial solution. After each irreducible graph computation we use the current best solution weight to prune the search tree. For this purpose we calculate an upper bound for the solution weight of the irreducible graph by using a clique cover heuristic and prune if we can't get a better current solution. If the irreducible graph consists of several components, we recursively calculate a solution on each component. Otherwise, a branching step is performed in which we branch into two subproblems. For this purpose, a static vertex order is computed at the beginning, which sorts vertices in non-decreasing order by their degree and breaks ties by their weight. We then branch using a case distinction for the currently highest vertex of this order.

## 4.1. Basic Reduction Algorithm

We now explain the reduction algorithm of the framework in detail. First, the reduction rules used are outlined and then the order in which they are applied is explained. In the following, a distinction is made between local and global reduction rules [33]. *Local reduction rules* only consider a small parts of the graph, which usually consist of a vertex and its neighborhood. *Global reduction rules* in contrast consider the entire graph and therefore usually take much more running time.

### 4.1.1. Reduction Rules

This section introduces the reduction rules used in the framework. The critical set reduction (Reduction Rule 9) is due to Butenko and Trukhanov [11] and is the only global reduction

**Algorithm 1** Branch-And-Reduce Framework

```
input graph G, current solution weight c (initially zero), best solution weight W (initially zero)
function Solve(G, c, W = 0)
    (G, c) ← Reduce(G, c)
    if W = 0 then W ← c + ILS(G)
    if c + UpperBound(G) ≤ W then return W
    if G is empty then return max(W, c)
    if G is not connected then
        for all G_i ∈ Components(G) do
            c ← c + Solve(G_i, 0, 0)
        return max(W, c)
    (G_1, c_1), (G_2, c_2) ← Branch(G, c)
    ▹ Run 1st case, update currently best solution
    W ← Solve(G_1, c_1, W)
    ▹ Use updated W to shrink the search space
    W ← Solve(G_2, c_2, W)
    return W
```

rule in the framework. The other reductions are local rules proposed by the authors Lamm et al. [33]. For more details and proofs we refer the reader to [11, 33].

**Local Reduction Rules.** Before we state the individual local reduction rules, we first introduce a basic graph operation called *vertex folding*. Let us assume a graph $G$ and a vertex $v$ with its neighborhood $N(v)$. We fold $v$ and $N(v)$ into a new vertex $v'$ by removing them from the graph and connecting $v'$ to each non-neighbor $u \in \overline{N}(v)$ which was previously adjacent to at least one of the vertices $w \in N(v)$, i.e $N(v') = N(N(v))$.

**Reduction Rule 1** (Neighorhood Removal)**.** *Let $v$ be a vertex with $w(v) \geq w(N(v))$. We obtain the transformed graph $G'$ by removing $N[v]$ from the graph. For an MWIS $I'$ of $G'$, the set $I = I' \cup \{v\}$ forms an MWIS of $G$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v)$.*

**Reduction Rule 2** (Degree Two Fold)**.** *Let $v$ be a vertex of degree two and its neighbors $u_1, u_2$ not adjacent. We also require $w(v) < w(u_1) + w(u_2)$ and $w(v) \geq \max\{w(u_1), w(u_2)\}$. $G'$ is obtained by folding $v, u_1, u_2$ into a single vertex $v'$ with weight $w(v') = w(u_1) + w(u_2) - w(v)$. For an MWIS $I'$ of $G'$ we construct an MWIS $I$ of $G$ in the following way: If $v' \in I'$ then we set $I = (I' \setminus \{v'\}) \cup \{u_1, u_2\}$, otherwise $I = I' \cup \{v\}$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v)$.*

**Reduction Rule 3** (Isolated Vertex Removal)**.** *Let $v$ be an isolated vertex, i.e. $G[N(v)]$ is a clique, with $w(v) \geq \max_{u \in N(v)} w(u)$. Then the transformed graph $G'$ is obtained by removal of the vertices $N[v]$. For an MWIS $I'$ of $G'$, the set $I = I' \cup \{v\}$ is an MWIS of $G$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v)$.*

**Reduction Rule 4** (Isolated Weight Transfer)**.** *Let $v$ be an isolated vertex, i.e. $G[N(v)]$ is a clique, and $S(v) \subseteq N(v)$ the set of isolated vertices $u \in N(v)$ with $w(v) \geq \max_{u \in S(v)} w(u)$.*

*We obtain the transformed graph $G'$ by removal of all vertices $u \in N(v)$ with $w(u) \leq w(v)$, lowering of the weights for all remaining vertices $x \in N(v)$ by $w(v)$, i.e. $w(x) = w(x) - w(v)$, and removal of the vertex $v$. For an MWIS $I'$ of $G'$ we construct an MWIS $I$ of $G$ in the following way: If $I' \cap N(v) = \emptyset$ then $I = I' \cup \{v\}$, otherwise $I = I'$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v)$.*

**Reduction Rule 5** (Domination). *Let $u, v$ be vertices such that $u$ is dominated by $v$, i.e. $N[u] \subseteq N[v]$, and $w(u) \geq w(v)$. The graph $G'$ is obtained by removal of $v$. For an MWIS $I'$ of $G'$, the set $I'$ also forms an MWIS of $G$. Furthermore we have $\alpha_w(G) = \alpha_w(G')$.*

**Reduction Rule 6** (Twin). *Let vertices $u, v$ be twins having independent neighborhoods $N(u) = N(v) = \{p, q, r\}$. Then we have two cases:*

1. *$w(\{u, v\}) \geq w(\{p, q, r\})$:*
   *The transformed graph $G'$ is obtained by removal of $u, v, p, q, r$. For an MWIS $I'$ of $G'$, the set $I = I' \cup \{u, v\}$ forms an MWIS of $G$.*

2. *$w(\{u, v\}) < w(\{p, q, r\})$ and $w(\{u, v\}) > w(\{p, q, r\}) - \min_{x \in \{p,q,r\}} w(x)$:*
   *The graph $G'$ is obtained by folding the vertices $u, v, p, q, r$ into a single vertex $v'$ with weight $w(v') = w(\{p, q, r\}) - w(\{u, v\})$. For an MWIS $I'$ of $G'$ we construct an MWIS $I$ of $G$ in the following way: If $v' \in I'$ then $I = (I' \setminus \{v'\}) \cup \{p, q, r\}$, otherwise $I = I' \cup \{u, v\}$.*

*Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(\{u, v\})$.*

**Reduction Rule 7** (Clique Neighborhood Removal). *Let $v$ be a vertex and $\mathcal{C}$ be a partition of its neighborhood $N(v)$ into cliques, i.e. any set $C \subset \mathcal{C}$ is a clique and $\bigcup_{C \in \mathcal{C}} C = V$. If furthermore $\sum_{C \in \mathcal{C}} \max_{u \in C} w(u) \leq w(v)$ holds, we obtain the transformed graph $G'$ by removing $N[v]$ from the graph. For an MWIS $I'$ of $G'$, the set $I = I' \cup \{v\}$ forms an MWIS of $G$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v)$.*

**Reduction Rule 8** (Generalized Fold). *Let $v$ be a vertex only having a single MWIS $I_{N(v)}$ with $w(I_{N(v)}) > w(v)$ in its induced neighborhood graph $G[N(v)]$. We obtain $G'$ by removal of any vertex $u \in N(v) \setminus I_{N(v)}$ and folding $I_{N(v)} \cup \{v\}$ into a single vertex $v'$ with weight $w(v') = w(I_{N(v)}) - w(v)$. For an MWIS $I'$ of $G'$ we construct an MWIS $I$ of $G$ in the following way: If $v' \in I'$ then $I = (I' \setminus \{v'\}) \cup I_{N(v)}$, otherwise $I = I' \cup \{v\}$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(\{u, v\})$.*

At this point we should note that the two reduction rules `Isolated Weight Transfer` and `Isolated Vertex Removal` are combined into one reduction rule in the framework, called `Clique Reduction`.

**Global Reduction Rules.** The only global reduction rule of the framework is based on critical independent sets. A subset $U_c \subseteq V$ is called a *critical weighted independent set* (CWIS) if $w(U_c) - w(N(U_c)) = \max\{w(U) - w(N(U) \mid U \subseteq V\}$ applies. Butenko and Trukanov [11] showed that each CWIS is always a subset of an MWIS. They calculate such a CWIS by solving the selection problem, which is equivalent to calculating a minimum cut [2]. Therefore this rule can be executed in polynomial time in the number of vertices.

**Algorithm 2** Incremental Reduction Rule Application

```
1:  input graph G, (zero indexed) reduction rule list R
2:  function IncrementalReduce(G,R)
3:      k ← 0                                    ▷ reduction rule selector
4:      while k < Length(R) do
5:          ▷ Try to reduce graph by kth reduction rule of R
6:          G′ ← ReduceByRule(R[k], G)
7:          if G′ = G then
8:              k ← k + 1
9:          else
10:             k ← 0
11:             G ← G′
12:     return G
```

**Reduction Rule 9** (Critical Weighted Independent Set)**.** *Let $U \subseteq V$ be a critical weighted independent set. We obtain the transformed graph $G'$ by removing $N[U]$. From an MWIS $I'$ in $G'$ we obtain an MWIS in $G$ using $I = I' \cup U$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(U)$.*

### 4.1.2. Reduction Rule Order

After the reduction rules have been presented in the previous section, we now explain in which order they are applied. For local reduction rules, the dependency checking method is presented first. It allows to apply reduction rules only to parts of the graph which have changed and can therefore be potentially reduced. Then we will briefly discuss the incremental reduction rule application, which provides a scheme for the order in which different reduction rules are applied.

#### 4.1.2.1. Dependency Checking

In general, local reduction rules are applied to a graph by attempting to execute them on any vertex in it. However, especially in later phases of the algorithm, local rules can only be applied to a few vertices of the graph. Thus, a complete check of the graph for each rule would require a lot of time for a minor decrease of the current graphs vertex number. The idea of dependency checking is to apply local rules only to vertices that can potentially be reduced. The basic observation is that after an unsuccessful application of a rule $R$ to a vertex $v$, the applicability of $R$ on $v$ only needs to be checked again when its neighborhood has changed. Therefore, for each local reduction rule $R$, a set $\mathcal{C}_R$ of all vertices which currently need to be checked is maintained. After a vertex has changed, we add it to $\mathcal{C}_R$ for each reduction rule, along with its neighborhood. Initially, these sets contain all vertices in the graph, so that for each vertex at least one attempt is made to apply each rule.

#### 4.1.2.2. Incremental Reductions

The idea of an iterative reduction rule application is to first reduce a graph by using reduction rules that are favorable in terms of running time and to switch to more expensive ones as soon as they are no longer applicable. At the beginning the reduction rules are therefore arranged in a list $\mathcal{R}$. The graph is then reduced by a current rule, which at the beginning is the first one in this list. If the graph has changed due to the successful application of the rule, we switch back to the first rule in the list. Otherwise we switch to the next rule in the list until the graph can no longer be reduced by any rule. This procedure is described in Algorithm 2. Overall, Lamm et al. apply their reduction rules in the following order: $\mathcal{R}$ = [Neighborhood Removal, Degree Two Fold, Clique Reduction, Domination, Twin, Clique Neighborhood Removal, Critical Weighted Independent Set, Generalized Fold].

# 5. Struction

In this chapter both the unweighted struction and the four different forms of the weighted struction are presented and illustrated with examples. We also provide correctness proves for each weighted variant for deeper understanding. Subsequently, for the weighted case, we examine relationships to other reduction rules that are already used in practice by outlining inclusions among them.

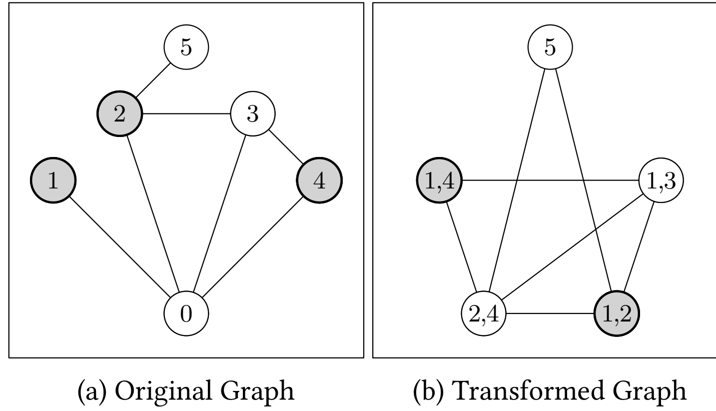
Before we introduce the individual struction variants, we first want to introduce some conventions and notation for this section. We call the vertex on which we want to apply one of the variants the *center vertex* and denote it by $v_0$. We arrange the vertices in its neighborhood $N(v)$ in an arbitrary but fixed order and index them according to their occurrence, i.e. $N(v_0) = \{v_1, v_2, ..., v_r\}$. For a (non-empty) subset $X \subseteq N(v_0)$ of neighbors, we denote the vertex with the smallest index of this order by $m(X)$ and the vertex with the largest index accordingly by $M(X)$. We denote the transformed graph by $G' = (V', E')$ and use the notation $N'$ and $w'$ for neighborhoods and vertex weights respectively.

Finally, both the unweighted and three of the four weighted variants use a construction where a set of vertices $U$ is partitioned into distinct *layers*. Therefore the set $U$ consists of vertices $v_{i,j}$, that are indexed by two parameters $i \in X, j \in Y$. The sets $X, Y$ either contain scalar values or vertex sets. For $k \in X$ a layer $L_k$ contains each vertex having $k$ as first parameter, i.e. $L_k = \{v_{i,j} \in U : i = k\}$. Conversely, we denote the *layer* of a vertex $v_{i,j}$ by $L(v_{i,j}) = i$.

## 5.1. Unweighted Struction

The main idea of the unweighted struction is to remove an arbitrary vertex $v_0$ and its neighbors from the graph and represent any independent sets in its neighborhood by a set of new vertices. In particular, an independent set $I_{N(v_0)} \subset N(v_0)$ is thereby represented by a set of vert pairs $(v_i, v_j)$ with $v_i = m(I_{N(v_o)})$. New vertices are therefore created for each non-adjacent neighbor pair $v_i, v_j \in N(v_0)$ with $i > j$. Since this representation requires one vertex less than in the original graph, we are able to reduce the stability number of the graph by one. In the following a formal definition of the transformation is given, an example can be found in Figure 5.1.

**Unweighted Reduction Rule 1** (Unweighted Struction). *Let $v_0 \in V$ be an arbitrary vertex. We derive the transformed graph $G'$ as follows: We remove $N[v_0]$ and create new vertices $v'_{i,j}$ for each pair of non-adjacent neighbors $v_i, v_j \in N(v_0)$ with $i < j$. We insert edges between two vertices $v'_{i_1,j_1}, v'_{i_2,j_2}$ if either $v_{j_1}$ and $v_{j_2}$ were adjacent or they belong to different layers, i.e. $i_1 \neq i_2$. Finally, each vertex $v'_{i,j}$ is also connected to each non-neighbor $v \in \overline{N}(v_0)$ adjacent to either $v_i$ or $v_j$. For an MIS $I'$ of $G'$ we obtain an MIS $I$ of $G$ as*




(a) Original Graph (b) Transformed Graph

Figure 5.1.: Application of unweighted struction. Vertices representing the same independent set in the different graphs are highlighted in gray. For simplicity we just write $i$ and $i, j$ instead of $v_i$ and $v_{i,j}$

*follows: If $I' \cap N(v_0) = \emptyset$ applies, we have $I = I' \cup \{v_0\}$, otherwise the new vertices in $I'$ are of the form $I' \setminus V = \{v'_{i,j_1}, v'_{i,j_2}, ..., v'_{i,j_r}\}$ and we replace them by the original vertices, i.e. $I = I' \cap V \cup \{v_i, v_{j_1}, v_{j_2}, ..., v_{j_r}\}$. Furthermore we have $\alpha(G) = \alpha(G') + 1$.*

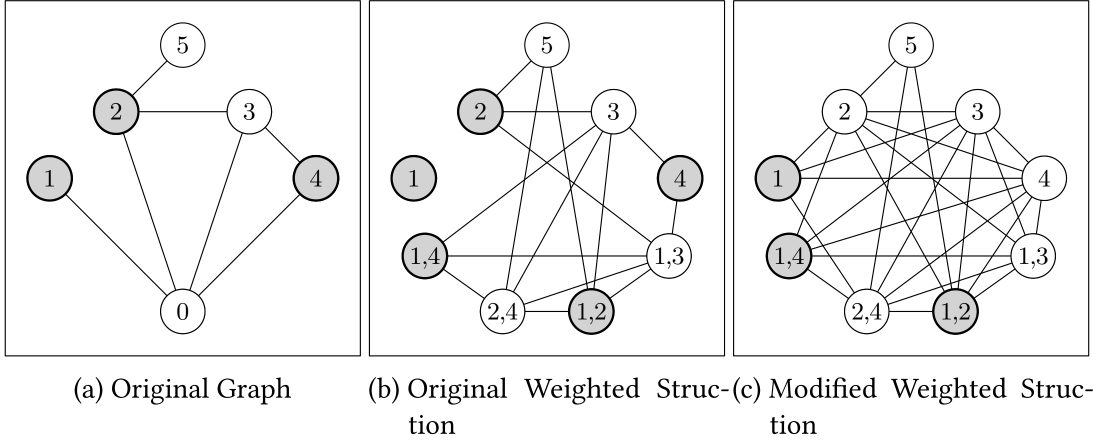
## 5.2. Weighted Struction Variants

The first presented struction variant is the original weighted struction introduced by Ebenegger et al. [19], while the other three variants presented are from Zavalnij [61].

### 5.2.1. Original Weighted Struction

In order to be applicable, in this variant the center vertex $v_0$ must have minimum weight among its neighbors, i.e. $w(v_0) < min_{v \in N(v_0)} w(v)$. In general, we apply the original weighted struction to the center vertex by removing $v_0$ and creating new vertices for each pair $v_i, v_j$ of non-adjacent vertices, i.e. an independent set of size two in the graph $G[N[v]]$. To guarantee that we can obtain an MWIS $I$ of $G$ by an MWIS $I'$ of $G'$ with $w(I) = w(I') + w(v_0)$, we also insert new edges between new and original vertices. In the following we describe the original weighted struction in detail. An example application of the original weighted struction can be found in Figure 5.2b.

**Reduction Rule 10** (Original Weighted Struction)**.** *Let $v_0 \in V$ be a vertex with minimum weight among its neighbors, i.e. $w(v_0) < min_{v \in N(v_0)} w(v)$. We derive the transformed graph $G'$ as follows: First, we remove $v_0$ and lower the weight of each neighbor $v_i \in N(v_0)$ by $w(v_0)$, i.e. $w'(v_i) = w(v_i) - w(v_0)$. For each pair of non-adjacent neighbors $v_i, v_j \in N(v_0)$ with $i < j$ we create a new vertex $v'_{i,j}$ with weight $w'(v'_{i,j}) = w(v_0)$. We insert edges between two vertices $v'_{i_1,j_1}, v'_{i_2,j_2}$ if either $v_{j_1}$ and $v_{j_2}$ are adjacent or they belong to different layers, i.e. $i_1 \neq i_2$. Finally, each vertex $v'_{i,j}$ is also connected to each vertex $v \in N(\{v_i, v_j\})$. For an MWIS $I'$ of $G'$ we obtain an MWIS $I$ of $G$ as follows: If $I' \cap N(v_0) = \emptyset$ applies, we*




(a) Original Graph (b) Original Weighted Struction (c) Modified Weighted Struction

Figure 5.2.: Application of original weighted struction and modified weighted struction. Vertices representing the same independent set in the different graphs are highlighted in gray. For simplicity we just write $i$ and $i, j$ instead of $v_i$ and $v_{i,j}$

*have* $I = I' \cup \{v_0\}$, *otherwise we remove the new vertices, i.e.* $I = I' \cap V$. *Furthermore we have* $\alpha_w(G) = \alpha_w(G') + w(v_0)$.

*Proof.* We show the statement by proving two sub-statements. First we state that for an MWIS $I'$ of $G'$ the obtained set $I$ forms an independent set of $G$ with $w(I) \geq w'(I') + w(v_0)$. Then we show how we can derive an independent set $I'_\star$ of $G'$ from an MWIS $I_\star$ of $G$ with $w(I_\star) = w'(I'_\star) + w(v_0)$. Finally we conclude that $\alpha_w(G) = \alpha_w(G') + w(v_0)$ holds and that $I$ is an MWIS of $G$.

To prove the first statement let $I'$ be an arbitrary MWIS of $G'$. First of all we can observe that for any vertex $v'_{i,j} \in I' \setminus V$ the vertices $v_i$ and $v_j$ are also contained in $I'$: Suppose an arbitrary vertex $v'_{i,j} \in I' \setminus V$. By construction, any neighbor $v \in N'(\{v_i, v_j\})$ is also a neighbor of $v'_{i,j}$. Since the vertices $v_i, v_j, v'_{i,j}$ are also pairwise non-adjacent, the set $I' \cup \{v_i, v_j\}$ forms an independent set of $G'$. We finally can conclude $v_i, v_j \in I'$ since $I'$ is an MWIS and the observation follows. We now distinguish two cases.
*Case 1*: $I' \cap N(v_0) = \emptyset$. Based on our observation, $I'$ only consists of non-neighbors of $v_0$, i.e. $I' \subseteq \overline{N}(v_0)$. Therefore $I = I' \cup \{v_0\}$ is an independent set in $G$ with weight $w(I) = w'(I') + w(v_0)$ .
*Case 2*: $I' \cap N(v_0) \neq \emptyset$. Since $I'$ is an independent set of $G'$, the set $I = I' \cap V$ forms an independent set of $G$. Furthermore, the vertices $I' \setminus V$ belong to the same layer, since by construction, new vertices of different layers are connected to each other. Following our initial observation, $I'$ contains less new vertices $V' \setminus V$ than neighbors $N(v_0)$ of $v_0$, i.e.

$$|I' \setminus V| \leq |I' \cap N(v_0)| + 1 .$$

Therefore we get the following relationship between vertex weights of the sets $I$ and $I'$

$$\begin{aligned}
w'(I' \setminus \overline{N}(v_0)) &= w'(I' \cap N(v_0)) + w'(I' \setminus V) \\
&= \sum_{v \in I' \cap N(v_0)} w'(v) + \sum_{v \in I' \setminus V} w'(v) \\
&= \sum_{v \in I' \cap N(v_0)} (w(v) - w(v_0)) + \sum_{v \in I' \setminus V} w(v_0) \\
&= \sum_{v \in I' \cap N(v_0)} w(v) - \sum_{v \in I' \cap N(v_0)} w(v_0) + \sum_{v \in I' \setminus V} w(v_0) \\
&= w(I' \cap N(v_0)) - |I' \cap N(v_0)| \cdot w(v_0) + |I' \setminus V| \cdot w(v_0) \\
&\leq w(I \setminus \overline{N}(v_0)) - w(v_0)\,,
\end{aligned}$$

that finally allows us to conclude

$$\begin{aligned}
w(I) &= w(I \cap \overline{N}(v_0)) + w(I \cap N(v_0)) \\
&\geq w'(I' \cap \overline{N}(v_0)) + w'(I' \setminus \overline{N}(v_0)) + w(v_0) \\
&= w'(I') + w(v_0)\,.
\end{aligned}$$

To show the second statement let $I_\star$ be an arbitrary MWIS of $G$. We also have two cases.
*Case 1*: $v_0 \in I_\star$. Since $I_\star$ is an independent set of $G$, the set $I'_\star = I_\star \setminus \{v_0\}$ forms an independent set of $G'$ with weight $w'(I'_\star) = w(I_\star) - w(v_0)$.
*Case 2*: $v_0 \notin I_\star$. Let $I_{N(v_0)} = I_\star \cap N(v_0)$ be the set of all neighbors $v_j \in N(v_0)$ contained in $I_\star$. This set is not empty, since otherwise $I_\star \cup \{v_0\}$ formed an independent with greater weight than $I$ and $I'$ was not an MWIS. Therefore let $v_i = m(I_{N(v_0)})$ be the vertex with minimum index in $I_{N(v_0)}$ and $I^-_{N(v_0)} = I_{N(v_0)} \setminus \{v_i\}$. We obtain $I'_\star$ by adding the vertex $v'_{i,j}$ to $I_\star$ for each vertex $v_j \in I^-_{N(v_0)}$, i.e $I'_\star = I_\star \cup \{v'_{i,j} \mid v_j \in I^-_{N(v_0)}\}$. Since the new vertices $I'_\star \setminus V$ belong to the same layer (and are therefore independent) and any vertex $v \in V$ that is adjacent to $v'_{i,j}$ is also adjacent to $v_i$ or $v_j$, we conclude that $I'_\star$ is independent. By construction of $I'_\star$, the number of new vertices $I'_\star \setminus V$ equals the number of neighbors $I'_\star \cap N(v_0)$ of $v_0$ plus one:

$$|I'_\star \setminus V| = |I'_\star \cap N(v_0)| + 1$$

and we therefore can derive a similar equation to case 2 of the first statement

$$w'(I'_\star \setminus \overline{N}(v_0)) = w(I_\star \setminus \overline{N}(v_0)) - w(v_0)\,,$$

which analogously gives us

$$w(I_\star) = w'(I'_\star) + w(v_0)\,.$$

Since $I'$ is an MWIS of $G'$ we can conclude by the two statements that

$$\begin{aligned}
w(I) &\geq w'(I') + w(v_0) \\
&\geq w'(I'_\star) + w(v_0) \\
&= w(I_\star)
\end{aligned}$$

applies. Since $I_\star$ is an MWIS of $G$, we therefore know that $I$ is also an MWIS of $G'$. Finally, we can conclude $\alpha_w(G) = \alpha_w(G') + w(v_0)$, since $I'$ and $I$ are maximum weight independent sets of $G'$ and $G$. □

### 5.2.2. Modified Weighted Struction

This variant extends the original weighted struction and aims to reduce the number of vertices in a maximum weight independent set in the transformed graph. While in the original weighted struction, the number of vertices of an MWIS increases in comparison to the original graph, this number remains the same in this variant. We therefore use a vertex construction quite similar to the original weighted struction that differs in the weight assignment of the new vertices and inserts some extra edges. An example application of the modified weighted struction can be found in Figure 5.2c.

**Reduction Rule 11** (Modified Struction). *Let $v_0 \in V$ be a vertex with minimum weight among its neighbors, i.e. $w(v_0) < \min_{v \in N(v_0)} w(v)$. We derive the transformed graph $G'$ as follows: First, we remove $v_0$ and lower the weight of each neighbor $v_i \in N(v_0)$ by $w(v_0)$, i.e. $w'(v_i) = w(v_i) - w(v_0)$. For each pair of non-adjacent neighbors $v_i, v_j \in N(v_0)$ with $i < j$ we create a new vertex $v'_{i,j}$ with weight $w'(v'_{i,j}) = w(v_j)$. We insert edges between two vertices $v'_{i_1,j_1}, v'_{i_2,j_2}$ if either $v_{j_1}$ and $v_{j_2}$ are adjacent or they belong to different layers, i.e. $i_1 \neq i_2$. Furthermore, we connect each vertex $v'_{i,j}$ to each non-neighbor $v \in \overline{N}(v_0)$ adjacent to either $v_i$ or $v_j$. We also connect each neighbor $v_k \in N(v_0)$ to each vertex $v'_{i,j}$ belonging to a different layer than k, i.e. $i \neq k$. Finally we extend $N(v_0)$ to a clique, i.e. we insert edges between vertices $v_i, v_j \in N(v_0)$ if they are not already present. For an MWIS $I'$ of $G'$ we obtain an MWIS $I$ of $G$ as follows: If $I' \cap N(v_0) = \emptyset$ applies, we have $I = I' \cup \{v_0\}$, otherwise we obtain $I$ by replacing each new vertex $v'_{i,j} \in I'$ with the original vertex $v_j$, i.e. $I = (I' \cap V) \cup \{v_j \mid v'_{i,j} \in I' \setminus V\}$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v_0)$.*

*Proof.* We show the statement by proving two sub-statements. First we state that for an MWIS $I'$ of $G'$ the obtained set $I$ forms an independent set of $G$ with $w(I) = w'(I') + w(v_0)$. Then we show how we can derive an independent set $I'_\star$ of $G'$ from an MWIS $I_\star$ of $G$ with $w(I_\star) = w'(I'_\star) + w(v_0)$. Finally we conclude that $\alpha_w(G) = \alpha_w(G') + w(v_0)$ holds and that $I$ is an MWIS of $G$. For the first statement we have two cases:
*Case 1*: $I' \cap N(v_0) = \emptyset$. Since $I'$ is an MWIS, we can show by contradiction that it does not contain any new vertex $v'_{i,j} \in V' \setminus V$, i.e. we have $I' \subseteq V$: Let us assume that there is a vertex $v'_{i,j} \in I' \setminus V$. Since every neighbor of $v_i$ in $G'$ is also a neighbor of $v'_{i,j}$, the set $I' \cup \{v_i\}$ forms an independent set of $G'$. As $I'$ is an MWIS, we therefore have $v_i \in I'$, which is a contradiction to the condition $I' \cap N(v_0) = \emptyset$. Therefore, the set $I = I' \cup \{v_0\}$ forms an independent set of $G$ with $w(I) = w'(I') + w(v_0)$.
*Case 2*: $I' \cap N(v_0) \neq \emptyset$. By construction of $G'$, for each vertex $v_j \in N(v_0)$ and $v'_{i,j} \in I' \setminus V$ we have $N(v_j) \subseteq N'(v'_{i,j})$. We therefore can obtain an independent set $I$ of $G$ by replacing each vertex $v'_{i,j} \in I' \setminus V$ with the original vertex $v_j$. Thus, $I = (I' \cap V) \cup I^-_{N(v_0)}$ with $I^-_{N(v_0)} = \{v_j \mid v'_{i,j} \in I' \setminus V\}$ forms an independent set of $G$. Since the vertices $v_j \in N(v_0)$ form a clique in $G'$, we have $I' \cap N(v_0) = \{v_i\}$. Furthermore, since any new vertex $v'_{k,j} \in V' \setminus V$ belonging to a layer $k \neq i$ is connected to $v_i$, the new vertices $I' \setminus V$ in $I'$ belong to the layer $i$. This leads us to the following equation:

$$w'(I' \setminus V) = w'(\{v'_{i,j_1}, v'_{i,j_2}, \ldots\}) = w(\{v_{j_1}, v_{j_2}, \ldots\}) = w(I^-_{N(v_0)})$$

and we can conclude

$$\begin{aligned} w(I) = w(I' \cap V) + w(I^-_{N(v_0)}) \\ &= w(I' \cap \overline{N}(v_0)) + w(v_i) + w(I^-_{N(v_0)}) \\ &= w'(I' \cap \overline{N}(v_0)) + w'(v_i) + w(v_0) + w'(I' \setminus V) \\ &= w'(I') + w(v_0)\ . \end{aligned}$$

To prove the second statement, let $I_\star$ be an arbitrary MWIS of $G$. We have two cases.
*Case 1*: $v_0 \in I_\star$. Since $I_\star$ is independent, $I'_\star = I_\star \setminus \{v_0\}$ is an independent set of $G'$ with $w(I_\star) = w'(I'_\star) - w(v_0)$.
*Case 2*: $v_0 \notin I_\star$. Let $I_{N(v_0)} = I_\star \cap N(v_0)$ be the set of all neighbors $N(v_0)$ contained in $I_\star$. This set is not empty, since otherwise $I_\star \cup \{v_0\}$ formed an independent set with greater weight than $I$ and $I'$ was not an MWIS. Therefore let $v_i = m(I_{N(v_0)})$ be the vertex with minimum index in $I_{N(v_0)}$ and $I^-_{N(v_0)} = I_{N(v_0)} \setminus \{v_i\}$. We obtain $I'_\star$ from $I_\star$ by replacing each vertex $v_j \in I'_{N(v_0)}$ with $v'_{i,j}$, i.e $I'_\star = (I_\star \setminus I^-_{N(v_0)}) \cup \{v'_{i,j} \mid v_j \in I^-_{N(v_0)}\}$. Since any new vertex $v'_{i,j} \in I'_\star \setminus V$ belongs to layer $i$ and $N'(v'_{i,j}) \cap \overline{N}(v_0) = N(v_i) \cup N(v_j)$ applies by construction of $G'$, we conclude that $I'_\star$ forms an independent set of $G'$. We now can show that the weight of the set $I'_\star$ meets the requirements as follows:

$$\begin{aligned} w'(I'_\star) &= w'(I_\star \setminus I^-_{N(v_0)}) + w'(\{v'_{i,j} \mid v_j \in I^-_{N(v_0)}\}) \\ &= w'(I_\star \setminus I^-_{N(v_0)}) + w(I^-_{N(v_0)}) \\ &= w'(I_\star \setminus I_{N(v_0)}) + w'(v_i) + w(I_{N(v_0)}) - w(v_i) \\ &= w(I_\star) + w'(v_i) - w(v_i) \\ &= w(I_\star) - w(v_0) \end{aligned}$$

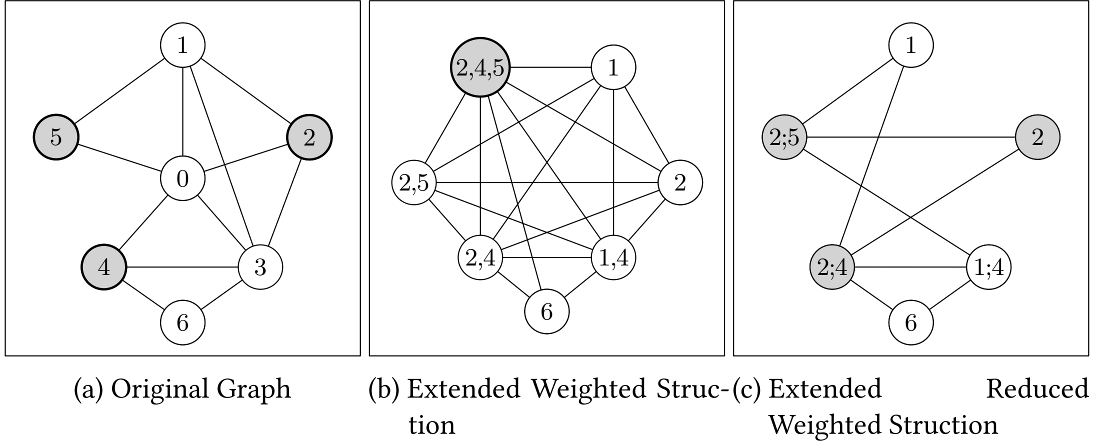
Since $I'$ is an MWIS of $G'$ we can conclude by the two statements that

$$\begin{aligned} w(I) &= w'(I') + w(v_0) \\ &\geq w'(I'_\star) + w(v_0) \\ &= w(I_\star) \end{aligned}$$

applies. Since $I_\star$ is an MWIS of $G$, we therefore know that $I$ is also an MWIS of $G'$. Finally, we can conclude $\alpha_w(G) = \alpha_w(G') + w(v_0)$, since $I'$ and $I$ are maximum weight independent sets of $G'$ and $G$. □

### 5.2.3. Extended Weighted Struction

The extended weighted struction removes the weight restriction for the vertex $v_0$ in the former variants. Unlike the previous two variants, this variant considers independent sets of arbitrary size in the neighborhood $N(v_0)$. In fact, we create new vertices for each independent set of $G[N(v)]$ if its weight is greater than $w(v_0)$. Since the number of independent sets $i(G)$ in an arbitrary graph $G$ of size $n$ can reach a maximum of $i(G) = 2^n$ [47], we consequently can create $\mathcal{O}(2^{\delta(v_0)})$ new vertices. An example application of the extended weighted struction can be found in Figure 5.3b.




(a) Original Graph (b) Extended Weighted Struction (c) Extended Reduced Weighted Struction

Figure 5.3.: Application of extended weighted struction and extended reduced weighted struction. Vertices representing the same independent set in the different graphs are highlighted in gray. We assume some weight constraints in the original graph for the construction in b) and c): $w(v_1) > w(v_0)$, $w(v_2) > w(v_0)$ and $w(v_3) + w(v_4) + w(v_5) \leq w(v_0)$. For simplicity we just write $i, .., j$ and $i, .., j; k$ instead of $v_{\{i,..,j\}}$ and $v_{\{i,..,j\},k}$.

**Reduction Rule 12** (Extended Weighted Struction). *Let $v_0 \in V$ be an arbitrary vertex and $\mathcal{C}$ the set of all independent sets $C$ of $G[N(v_0)]$ with $w(C) > w(v_0)$. We derive the transformed graph $G'$ as follows: First, we remove the vertices $N[v_0]$ and create a new vertex $v'_c$ with weight $w'(v'_c) = w(c) - w(v_0)$ for each independent set $c \in \mathcal{C}$. We connect each vertex $v'_c$ to each vertex $v \in \overline{N}(v_0) \cap N(c)$. Finally, the vertices $v'_c$ are connected with each other, forming a clique. For an MWIS $I'$ of $G'$ we obtain an MWIS $I$ of $G$ as follows: If $I' \setminus V = \emptyset$ applies, we set $I = I' \cup \{v_0\}$, otherwise there is a single vertex $v'_c \in I' \setminus V$ that we replace with the vertices of its independent set $c$, i.e. $I = (I' \cap V) \cup c$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v_0)$.*

*Proof.* We show the statement by proving two sub-statements. First we state that for an MWIS $I'$ of $G'$ the obtained set $I$ forms an independent set of $G$ with $w(I) = w'(I') + w(v_0)$. Then we show how we can derive an independent set $I'_\star$ of $G'$ from an MWIS $I_\star$ of $G$ with $w(I_\star) = w'(I'_\star) + w(v_0)$. Finally we conclude that $\alpha_w(G) = \alpha_w(G') + w(v_0)$ holds and that $I$ is an MWIS of $G$. For the first statement we have two cases:
*Case 1*: $I' \setminus V = \emptyset$. Since $I'$ is an independent set of $G'$, the set $I = I' \cup \{v_0\}$ forms an independent set of $G$ with $w(I) = w'(I') + w(v_0)$.
*Case 2*: $I' \setminus V \neq \emptyset$: Since the new vertices $V' \setminus V$ form a clique, we have $I' \setminus V = \{v'_c\}$. By the construction of $G'$, the set $c$ is independent and for each vertex $v \in c$ we have $N(v) \cap \overline{N}(v_0) \subseteq N'(v'_c)$. Therefore the set $I = I' \setminus \{v'_c\} \cup c$ forms an in-

dependent set in $G$ with weight

$$\begin{aligned} w(I) &= w(I' \setminus \{v'_c\}) + w(c) \\ &= w'(I' \setminus \{v'_c\}) + w'(v'_c) + w(v_0) \\ &= w'(I') + w(v_0) . \end{aligned}$$

To prove the second statement, let $I_\star$ be an arbitrary MWIS of $G$. Again we have two cases:
*Case 1*: $v_0 \in I_\star$. Since $I_\star$ is an independent set, we have $I_\star \cap N(v_0) = \emptyset$. Therefore $I'_\star = I_\star \setminus \{v_0\}$ is an independent set of $G'$ with $w(I_\star) = w(I'_\star) - w(v_0)$.
*Case 2*: $v_0 \notin I_\star$. Let $I_{N(v_0)} = I_\star \cap N(v_0)$ be the set of all neighbors $N(v_0)$ contained in $I_\star$. If $w(I_{N(v_0)}) \leq w(v_0)$ applies, the set $(I_\star \setminus I_{N(v_0)}) \cup \{v_0\}$ forms an independent set of $G$ with weight

$$\begin{aligned} w((I_\star \setminus I_{N(v_0)}) \cup \{v_0\}) &= w(I_\star) - w(N(v_0)) + w(v_0) \\ &\geq w(I_\star) , \end{aligned}$$

which we can handle by case 1. Therefore, w.l.o.g. we can assume $w(I_{N(v_0)}) > w(v_0)$, resulting in the existence of a vertex $v'_{I_{N(v_0)}} \in V' \setminus V$. Since $I_\star$ is an independent set of $G$ and $N'(v'_{I_{N(v_0)}}) = N(I_{N(v_0)}) \cap \overline{N}(v_0)$ holds, we can replace the vertices $I_{N(v_0)}$ with $v'_{I_{N(v_0)}}$ and obtain an independent set $I'_\star = (I_\star \setminus I_{N(v_0)}) \cup \{v'_{I_{N(v_0)}}\}$ which satisfies

$$\begin{aligned} w'(I'_\star) &= w'(I_\star \setminus I_{N(v_0)}) + w'(v'_{I_{N(v_0)}}) \\ &= w(I_\star \setminus I_{N(v_0)}) + w(I_{N(v_0)}) - w(v_0) \\ &= w(I_\star) - w(v_0) . \end{aligned}$$

Since $I'$ is an MWIS of $G'$ we can conclude by the two statements that

$$\begin{aligned} w(I) &= w'(I') + w(v_0) \\ &\geq w'(I'_\star) + w(v_0) \\ &= w(I_\star) \end{aligned}$$

applies. Since $I_\star$ is an MWIS of $G$, we therefore know that $I$ is also an MWIS of $G'$. Finally, we can conclude $\alpha_w(G) = \alpha_w(G') + w(v_0)$, since $I'$ and $I$ are maximum weight independent sets of $G'$ and $G$. □

### 5.2.4. Extended Reduced Weighted Struction

The extended reduced weighted struction is a modification of the extended weighted struction and aims to reduce the number of new vertices. While the extended weighted struction considers all independent sets $C$ in $G[N(v_0)]$ whose weight is greater than $w(v_0)$, in this variant we only look at a subset $C' \subseteq C$ of the "just" greater independent sets. Such a set $c \in C'$ is characterized by having a weight greater than $w(v_0)$, but having a smaller or equal weight than $v_0$ without its vertex with the largest index $M(c)$. The idea of

the extended reduced weighted struction is that any other set $c \in C \setminus C'$ can be obtained from a set $c' \in C'$ by expanding it with additional vertices. For this purpose, the same construction as in the extended weighted struction is applied to the set $C'$ instead of $C$. Moreover, we add additional vertices which serve as an extension for a set $c' \in C'$. In total, this variant creates at most as many vertices as its predecessor, but can also create fewer vertices. An example for the latter case is illustrated in Figure 5.3c.

**Reduction Rule 13** (Extended Reduced Weighted Struction)**.** *Let $v_0 \in V$ be an arbitrary vertex, $C$ be the set of all independent sets $c$ of $G[N(v_0)]$ and $C'$ be the subset of "just" greater independent sets, i.e. $C' = \{c \in C \mid w(c) - w(M(c)) \leq w(v_0)\}$. We derive the transformed graph $G'$ as follows: We remove the vertices $N[v_0]$ and create a new vertex $v'_c$ with weight $w'(v'_c) = w(c) - w(v_0)$ for each independent set $c \in C'$. We denote the set of these vertices $v'_c$ by $V_C$. We connect any vertex $v'_c$ to each vertex $v \in \overline{N}(v_0) \cap N(c)$. The vertices $v'_c$ are also connected with each other, forming a clique. For each pair consisting of an independent set $c \in C'$ and a vertex $v_j \in N(v_0)$ we create a vertex $v'_{c,j}$ with weight $w'(v'_{c,j}) = w(v_j)$, if $c$ can be extended by $v_j$, i.e. $v$ is not adjacent to any vertex $v' \in c$. We denote the set of these vertices $v'_{c,j}$ by $V_E$. We insert edges between two vertices $v'_{c_1,j_1}, v'_{c_2,j_2}$ if they either belong to different layers, i.e. $c_1 \neq c_2$, or $v_{j_1}$ and $v_{j_2}$ have been adjacent. Moreover, we connect any vertex $v'_{c,j}$ to each vertex $v \in \overline{N}(v_0) \cap N(c \cup \{v_j\})$. Finally we connect each vertex $v'_{c_1,j}$ to each vertex $v'_{c_2}$ belonging to a different layer than $c_1$, i.e. $c_1 \neq c_2$. For an MWIS $I'$ of $G'$ we obtain an MWIS $I$ of $G$ as follows: If $I' \cap V_C = \emptyset$ applies, we set $I = I' \cup \{v_0\}$. Otherwise, there is a single vertex $v'_c \in I' \cap V_C$ that we replace with the vertices of its independent set $c$. Moreover, we replace each vertex $v'_{c,j} \in I' \cap V_E$ with the vertex $v_j$. Altogether we have $I = (I' \cap V) \cup c \cup \{v_j \mid v'_{c,j} \in I' \cap V_E\}$. Furthermore we have $\alpha_w(G) = \alpha_w(G') + w(v_0)$.*

*Proof.* We show the statement by proving two sub-statements. First we state that for an MWIS $I'$ of $G'$ the obtained set $I$ forms an independent set of $G$ with $w(I) = w'(I') + w(v_0)$. Then we show how we can derive an independent set $I'_\star$ of $G'$ from an MWIS $I_\star$ of $G$ with $w(I_\star) = w'(I'_\star) + w(v_0)$. Finally we conclude that $\alpha_w(G) = \alpha_w(G') + w(v_0)$ holds and that $I$ is an MWIS of $G$. For the first statement we have two cases:
*Case 1*: $I' \cap V_C = \emptyset$. We first show by contradiction that $I'$ only consists of vertices $v \in V$, i.e. $I' \subseteq V$ applies: Since we have $I' \cap V_C = \emptyset$, let us assume that there is a vertex $v'_{c,j} \in I' \cap V_E$. By construction, each neighbor of the vertex $v'_c$ is also a neighbor of the vertex $v'_{c,j}$. Thus, the set $I' \cup \{v'_c\}$ forms an independent set of $G'$. By our assumption we have $I' \cap V_C$ and therefore $v'_c \notin I'$ applies. This finally is a contradiction to the assumption that $I'$ is an MWIS, since we have $w(I' \cup \{v'_c\}) > w(I')$. Therefore, the set $I = I' \cup \{v_0\}$ forms an independent set of $G$ with $w(I) = w(I') + w(v_0)$.
*Case 2*: $I' \cap V \neq \emptyset$: Since the vertices $V_C$ form a clique, there is a unique vertex $v'_c \in V_C$ that satisfies $I' \cap V = \{v'_c\}$. Moreover, since any vertex $v'_{c',j} \in V_E$ is connected to $v'_c$, if $c' \neq c$ applies, the vertices $I' \cap V_E$ belong to the same layer $c$.
First of all, in the following we show that the set $I = (I' \cap V) \cup c \cup I^-_{N(v_0)}$ with $I^-_{N(v_0)} = \{v_j \mid v'_{c,j} \in I' \cap V_E\}$ forms an independent set in $G$: By construction, each vertex $v'_{c,j} \in V' \setminus V$ is adjacent to each neighbor $v \in N(v_j)$, i.e. $N'(v'_{c,j}) \subseteq N(v_j)$ applies. Therefore we can replace each vertex $v'_{c,j} \in I' \cap V_E$ with $v_j$ and obtain an independent set

in $G'$ by $(I' \cap V) \cup I^-_{N(v_0)}$. Moreover, since the the vertex $v'_c$ is created by an independent set $c$ and we have $N'(v'_{c,j}) = N(c) \cap \overline{N}(v_0)$, the set $(I' \cap V) \cup c$ is also independent. Finally, we can state per proof by contradiction that $I = (I' \cap V) \cup c \cup I^-_{N(v_0)}$ forms an independent set of $G$: Suppose $I$ is not an independent set of $G$. Since $(I' \cap V) \cup I^-_{N(v_0)}$ and $(I' \cap V) \cup c$ are independent, there are vertices $v \in c$ and $v_j \in I^-_{N(v_0)}$ with $\{v, v_j\} \in E$. Consequently, the set $c \cup \{v_j\}$ is not independent and we have $v'_{c,j} \notin V'$. However, since all vertices $I' \cap V_E$ belong to the same layer $c$, we have $v_j \notin I^-_{N(v_0)}$, which is a contradiction to the assumption $v_j \in I^-_{N(v_0)}$. Therefore $I$ forms an independent set of $G$ and we are left to show that $w(I) = w'(I') + w(v_0)$ is satisfied.
Since the vertices $I' \cap V_E$ belong to the same layer we have the following equation:

$$w'(I' \cap V_E) = w'(\{v'_{c,j_1}, v'_{c,j_2}, \ldots\}) = w(\{v_{j_1}, v_{j_2}, \ldots\}) = w(I^-_{N(v_0)}) \,.$$

Moreover we have $w(I' \cap V) = w'(I' \cap V)$ and $w(c) = w'(v'_c) + w(v_0)$, leading us to the conclusion

$$\begin{aligned} w(I) &= w(I' \cap V) + w(c) + w(I^-_{N(v_0)}) \\ &= w'(I' \cap V) + w'(v'_c) + w(v_0) + w'(I' \cap V_E) \\ &= w'(I' \cap V) + w'(I' \cap V_C) + w'(I' \cap V_E) + w(v_0) \\ &= w'(I') + w(v_0) \,. \end{aligned}$$

To prove the second statement, let $I_\star$ be an arbitrary MWIS of $G$. We have two cases.
*Case 1*: $v_0 \in I_\star$. Since $I_\star$ is independent, $I'_\star = I_\star \setminus \{v_0\}$ is an independent set of $G'$ with $w(I_\star) = w'(I'_\star) - w(v_0)$.
*Case 2*: $v_0 \notin I_\star$. Let $I_{N(v_0)} = I_\star \cap N(v_0)$ be the set of all neighbors $N(v_0)$ contained in $I_\star$. Based on the order of the neighbors $N(v_0) = \{v_1, v_2, \ldots, v_r\}$ we denote this set by $I_{N(v_0)} = \{v_{j_1}, v_{j_2}, \ldots, v_{j_s}\}$ with $j_1 < j_2 < \ldots < j_s$. If $w(I_{N(v_0)}) \leq w(v_0)$ applies, the set $(I_\star \setminus I_{N(v_0)}) \cup \{v_0\}$ forms an independent set of $G$ with weight

$$\begin{aligned} w((I_\star \setminus I_{N(v_0)}) \cup \{v_0\}) &= w(I_\star) - w(N(v_0)) + w(v_0) \\ &\geq w(I_\star) \,, \end{aligned}$$

which we can handle by case 1. Therefore, w.l.o.g. we can assume $w(I_{N(v_0)}) > w(v_0)$. For this reason, there is a $K \in [1, s]$, so that the set $c = \bigcup_{k \leq K} v_{j_k}$ fulfills both $w(c) > w(v_0)$ and $w(c) - w(M(c)) \leq w(v_0)$. Consequently there is a vertex $v'_c \in V_C$. Furthermore, let $I^-_{N(v_0)} = I_{N(v_0)} \setminus c$ be the set of vertices $v \in I_{N(v_0)}$ that are not contained in $c$. We now construct an independent set $I'_\star$ of $G'$ from $I_\star$ by removing the vertices $c$, adding the vertex $v'_c$ and replacing each vertex $v_j \in I_{N(v_0)}$ with $v'_{c,j}$. Thus, we have $I'_\star = (I_\star \cap \overline{N}(v_0)) \cup \{v'_c\} \cup \{v'_{c,j} \mid v_j \in I_{N(v_0)}\}$. Since we have $N'(v'_c) \subseteq N(c)$, all vertices $I'_\star \cap V_E$ belong to the same layer $c$ and $N'(v'_{c,j}) \cap \overline{N}(v_0) = N(c \cup \{v_j\})$ holds, the set $I'_\star$ is an independent set of $G'$. We now can show that the weight of the set $I'_\star$ meets

the requirements as follows:

$$\begin{aligned} w'(I'_\star) &= w'(I_\star \cap \overline{N}(v_0)) + w'(v'_c) + w'(\{v'_{i,j} \mid v_j \in I^-_{N(v_0)}\}) \\ &= w'(I_\star \cap \overline{N}(v_0)) + w'(v'_c) + w(I^-_{N(v_0)}) \\ &= w(I_\star \cap \overline{N}(v_0)) + w(c) - w(v_0) + w(I^-_{N(v_0)}) \\ &= w(I_\star \cap \overline{N}(v_0)) + w(I_\star \cap N(v_0)) - w(v_0) \\ &= w(I_\star) - w(v_0) \end{aligned}$$

Since $I'$ is an MWIS of $G'$ we can conclude by the two statements that

$$\begin{aligned} w(I) &= w'(I') + w(v_0) \\ &\geq w'(I'_\star) + w(v_0) \\ &= w(I_\star) \end{aligned}$$

applies. Since $I_\star$ is an MWIS of $G$, we therefore know that $I$ is also an MWIS of $G'$. Finally, we can conclude $\alpha_w(G) = \alpha_w(G') + w(v_0)$, since $I'$ and $I$ are maximum weight independent sets of $G'$ and $G$. □

## 5.3. Relationship To Other Reduction Rules

After we have presented reduction rules that have already been used in practice in the previous chapter, in this section we want to identify relationships between them and the different variants of the struction. The aim of this section is to show which of these rules are already subsumed by struction. We say that a reduction rule $R_1$ is *subsumed* by a reduction rule $R_2$ if for each graph $G_1$ produced by any application of $R_1$ we can obtain an equivalent graph $G_2$ by the application of $R_2$. For the inclusion of a reduction rule of $R_1$ by $R_2$ we write $R_1 \subseteq R_2$. To show the different inclusions to existing rules, we always use the special case where the extended weighted struction and extended reduced weighted struction perform the same graph transformation. This is the case for a vertex $v_0$, if the set of the just greater independent sets $C'$ is equal to the set of all independent sets $C$ in $G[N(v_0)]$ with greater weight than $w(v_0)$. We refer to this special case in the following as Extended Weighted Struction$_{C=C'}$. An overview on existing inclusions is given in Figure 5.4.

### 5.3.1. Clique Neighborhood Removal ⊆ Extended Weighted Struction$_{C=C'}$

Let $v$ be a vertex on which the clique neighborhood reduction rule can be applied and $G'$ be the transformed graph obtained by removal of $N[v]$. Since there exists a partition $\mathcal{C}$ of $N[v]$ into cliques with $w(v) \geq \sum_{C \in \mathcal{C}} \max_{u \in C} w(u)$ and any independent set of $G$ can include at most one vertex of each clique, there is no independent set $I_{N(v)}$ in $G[N(v)]$ with $w(I_{N(v)}) \geq w(v)$ and we therefore have $C = C' = \emptyset$. Thus, when applying the extended weighted struction to $v$, we remove $N[v]$ and do not insert any new vertex, resulting in the same graph $G'' = G'$.

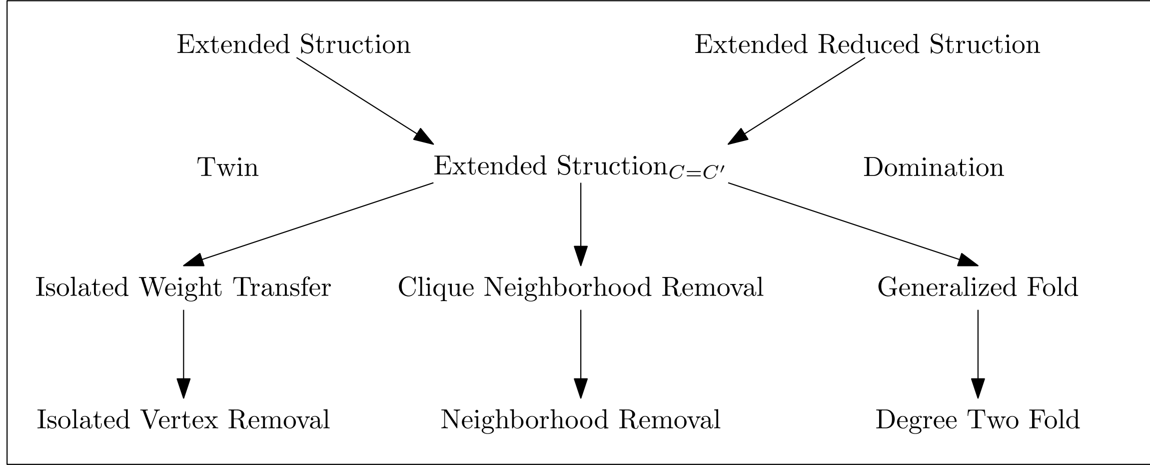


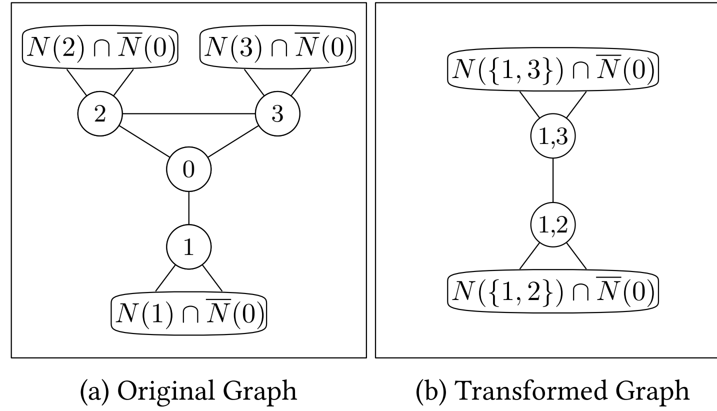

Figure 5.4.: Relationships between reduction rules. An inclusion $R_1 \subseteq R_2$ is visualized by an arrow from $R_2$ to $R_1$.

### 5.3.2. Generalized Fold $\subseteq$ Extended Weighted Struction$_{C=C'}$

Let $v$ be a vertex on which the generalized fold rule is applicable and $G'$ be the transformed graph. Due to the preconditions there is exactly one MWIS $I_{N(v)}$ with $w(I_{N(v)}) > w(v)$ in $G[N(v)]$ and we therefore have $C = C' = \{I_{N(v)}\}$. Thus, when we apply the extended weighted struction on $v$, we remove the vertices $N[v]$ and insert a single new vertex $v'_{I_{N(v)}}$. By construction the neighborhood of this vertex is $N(v'_{I_{N(v)}}) = N(I_{N(v)})$, which corresponds to a fold of the vertices $N[v]$. Moreover, our vertex $v'_{I_{N(v)}}$ and $v'$, the one obtained in the generalized fold reduction, have the same weight, because $w'(v'_{I_{N(v)}}) = w(I_{N(v)}) - w(v) = w'(v')$ holds. Thus we obtain the same graph $G'' = G'$.

### 5.3.3. Isolated Weight Transfer $\subseteq$ Extended Weighted Struction$_{C=C'}$

Let $v$ be a vertex on which the isolated weight transfer rule is applicable and $G' \setminus N(v)]$ be the transformed graph. Furthermore, let $S(v) \subseteq N(v)$ be the set of all isolated vertices in $N(v)$. Since $v$ is an isolated vertex, its neighborhood $N(v)$ forms a clique. Each independent set $I_{N(v)}$ of $G[N(v)]$ thus consists of a single vertex $u \in N(v)$ and we have $C = C'$. Therefore, when we apply the extended weighted struction to $v$ we create a new vertex $v'_{\{u\}}$ if and only if $w(u) > w(v)$ applies, i.e. if we do not remove $u$ from the graph during the application of the isolated weight transfer rule. We now state that both transformed graphs are equivalent by showing that these vertices $v'_{\{u\}}$ and $u$ are equivalent. By construction we have $w'(v'_{\{u\}}) = w(\{u\}) - w(v) = w(u) - w(v) = w'(u)$, so they fulfill the weight condition. In addition, $N(v'_{\{u\}}) \cap \overline{N}(v) = N(u) \cap \overline{N}(v)$ applies, since we connect $v'_{\{u\}}$ to each non-neighbor $x \in \overline{N}(v)$ being adjacent to $u$. Finally, both the remaining vertices $u \in N(v)$ and the new vertices $v'_{\{u\}}$ form a clique, resulting in the equivalence of the neighborhoods of $u$ and $v'_{\{u\}}$.




(a) Original Graph (b) Transformed Graph

Figure 5.5.: Application of extended (reduced) weighted struction on simple new reducible graph structure. For simplicity we just write $i$ and $i, j$ instead of $v_i$ and $v_{i,j}$

### 5.3.4. Weighted Isolated Vertex Removal ⊆ Isolated Weight Transfer

Let $v$ be a vertex on which the weighted isolated vertex removal rule is applicable. Then $w(v) \geq \max_{u \in N(v)} w(u)$ holds, implying the condition $w(v) \geq \max_{u \in S(v)} w(u)$ for the set of all isolated vertices $S(v) \subseteq N(v)$. Thus, the isolated weight transfer rule is applicable to $v$ and removes the vertices $N[v]$ from the graph, giving us the same graph $G'' = G'$.

### 5.3.5. Degree Two Fold ⊆ Generalized Fold

Let $v$ be a vertex on which the degree two fold reduction is applicable and $G'$ be the transformed graph obtained by folding $N[v]$ into a single vertex. According to the precondition, $v$ has an independent neighborhood $N(v) = \{u_1, u_2\}$ with $w(v) < w(u_1) + w(u_2)$ and $w(v) \geq \max\{w(u_1), w(u_2)\}$. Therefore there only exists exactly one independent set $I_{N(v)} = \{u_1, u_2\}$ with $w(I_{N(v)}) > w(v)$ of $G[N(v)]$. Consequently, the generalized fold rule can be applied to $v$ and creates the same graph $G'' = G'$, since it also folds the vertices $N[v]$ into a new vertex.

### 5.3.6. Neighborhood Removal ⊆ Clique Neighborhood Removal

Let $v$ be a vertex on which the neighborhood removal rule can be applied and $G'$ be the transformed graph constructed by removal of $N[v]$. We can obtain a partition of $N(v)$ into cliques by cliques of size one, i.e. $\mathcal{C} = \bigcup_{u \in N(v)} \{u\}$. Since we have $w(v) \geq w(N(v))$, we can conclude $\sum_{C \in \mathcal{C}} \max_{u \in C} w(u) = w(N(v)) \leq w(v)$. Therefore the Clique Neighborhood Removal reduction is applicable on $v$ and we obtain the same graph $G'' = G'$, since we also remove $N[v]$ from the graph.

### 5.3.7. New Reducible Graph Structures

Finally, we want to show briefly which previously irreducible graph structures can be reduced by the extended (reduced) weighted struction in such a way that the transformed graph has fewer nodes than before. In general these consist of vertices whose neighborhood $N(v)$ does not form a clique and has at least two independent sets of greater weight than $w(v)$ and there are no twin or domination relationships of vertices $v \in N[v]$. Such graph structures can occur even with nodes of small degree, an example of a degree three vertex is shown in Figure 5.5b.

# 6. Efficient Data Reduction Via Struction

In this chapter, the weighted struction is integrated into the branch-and-reduce framework presented in the previous chapter and two new preprocessing algorithms are developed. Our general goal in data reduction is to transform an input graph to a simpler, smaller graph. Since all struction variants are very general reduction rules, which do not necessarily reduce the number of vertices but can create a large number of new vertices, special cases of structions are considered. For this purpose, reduction rules are first divided into three different types, depending on how the size of the original graph and transformed graph differ.

For *decreasing reduction rules* or short *decreasing reductions* the transformed graph $G'$ has less vertices than the original graph $G$. All reduction rules in the basic reduction algorithm of Lamm et al. are of this type. We derive special types of struction, which also belong to this type. Reduction rules where the number of vertices in the original graph is the same as in the transformed graph are called *plateau reductions*. The graphs $G'$ and $G$ differ only structurally, the vertices have different adjacencies. While plateau reductions cannot reduce the size of a graph, they can potentially produce new subgraphs which can then be reduced by other (decreasing) reductions. Finally, *increasing reductions* denote those reduction rules whose transformed graph has more vertices than the original graph. Similar to plateau reductions, the idea is to reduce the resulting graph by further reduction rules. However, increasing structions can lead to a general growth of the graph, even if it can be reduced again after its application. Therefore, it is difficult to integrate them into an incremental reduction algorithm.

Another way we restrict the struction in practice is to define a maximum vertex degree $d_{max}$ up to which a struction is applicable. Since the number of new nodes can grow asymptotically exponentially in the vertex degree, we try to avoid applicability checks as well as the actual execution of expensive structions.

In the next section, our first algorithm is presented, called non-increasing reduction algorithm, which extends the basic reduction algorithm of Section 4.1 by further struction based reduction rules. The existing reduction rules are therefore extended by decreasing and plateau structions. These are very general rules, which already cover many rules of the framework and can also reduce some subgraphs, which are irreducible for the existing rules. Overall, we hope to be able to obtain smaller irreducible graphs on different instances. Afterwards we present our second algorithm, called cyclic blow-up algorithm, which extends the non-increasing reduction algorithm by using increasing structions. Intuitively seen, this extends the reduction space even more, so that potentially even smaller irreducible graphs can be calculated. However, increasing structions can lead to a large increase in the number of vertices of the graph depending on the choice, so special techniques are needed to avoid this problem.

## 6.1. Non-Increasing Reduction Algorithm

In this section we show how to obtain decreasing and plateau structions from the four different struction variants. Based on this, an incremental reduction algorithm is developed, which extends the basic reduction algorithm of Lamm et al.by these struction types.

In general, when applying any struction variant, the number of vertices of the transformed graph $G'$ depends on the number of removed and newly created vertices. If more vertices are removed than newly created, we are dealing with a decreasing struction. If both numbers are equal, we have a plateau struction. It is difficult to estimate the number of newly created vertices in advance, since all variants depend on certain independent sets in the neighborhood $N(v)$ of the center vertex. The general method is therefore to use one of the struction variants and count the number of vertices created during this process. If these exceed a maximum value $n_{max}$, the execution of the struction is aborted.

At first we have a look at the structurally very similar variants original weighted struction and modified weighted struction. These can reduce the number of vertices in the graph by a maximum of one, since they only remove the center vertex $v$. Decreasing or plateau structions can be found by attempting to execute one of the two struction variants with $n_{max} = 0$ or $n_{max} = 1$. At this point, however, we can see that an decreasing struction is already covered by the isolated weight transfer reduction of the framework: In both struction variants a new vertex is created for each non-adjacent vertex pair $v_i, v_j \in N(v)$ in the neighborhood of $v$. Consequently, no new vertex is created exactly when the neighborhood $N(v)$ forms a clique. Thus $v$ is isolated and is removed when the isolated weight transfer rule is executed on $v$ or a neighbor vertex $u \in N(v)$. This means that we cannot derive new reduction rules for these two struction variants that reduce the number of vertices. A plateau struction occurs if exactly one pair of vertices $v_i, v_j \in N(v)$ exists, which are not adjacent to each other.

This is different for the other two struction variants. Since they remove the center vertex $v$ and its neighborhood $N(v)$, the graph size can be reduced by up to $\delta(v) + 1$. Decreasing or plateau structions can be found by executing the corresponding struction variant with $n_{max} = \delta(v)$ or $n_{max} = \delta(v) + 1$.

Finally, we show how to integrate the new rules into the basic reduction algorithm from Section 4.1. To a large extent, we take the reduction rules and their sequence from the basic reduction algorithm. At this point we decide to only use one of the four struction types simultaneously. This allows us to better determine the effectiveness of each individual struction variant in the later evaluation. Since the last two struction types in particular are very general reduction rules, they tend to be expensive in terms of runtime. We therefore apply them as the last local reduction rule before we move on to the global critical set reduction. Overall, we apply the reduction rules in the following order: $R$ = [Neighborhood Removal, Degree Two Fold, Clique Reduction, Domination, Twin, Clique Neighborhood Removal, Generalized Fold, Decreasing Struction, Plateau Struction, Critical Weighted Independent Set].

In Section 5.3 we have already seen that the extended weighted struction and the extended reduced weighted struction subsume several reduction rules. Therefore, in the later evaluation we will examine whether we can obtain advantages in terms of runtime by only using a subset of these reduction rules.

**Algorithm 3** Cyclic Blow-Up Algorithm

```
1: input graph G
2: function CyclicBlowUp(G)
3:     K ← Reduce(G)                                        ▹ current graph
4:     K* ← K                                               ▹ best graph
5:     count ← 0                                            ▹ iterations since last accept
6:     while Vertices(K) < α · Vertices(K*) and count < X do
7:         ▹ Modify current graph structure by application of increasing structions
8:         K′ ← BlowUp(K)
9:         if K′ = K then
10:            return K*
11:        ▹ Shrink graph size by non-increasing reduction application
12:        K″ ← Reduce(K′)
13:        ▹ Accept or reject new graph K″ for next iteration
14:        (K, K*, count) ← Accept(K″, K, K*, count)
15:    return K*
```

## 6.2. Cyclic Blow-Up Algorithm

In this section we introduce another reduction algorithm, called cyclic blow-up algorithm, which is an extension of the non-increasing reduction algorithm from the previous section. In the branch-and-reduce framework, this algorithm can theoretically be used whenever an irreducible graph is calculated, i.e. before each branching step. In practice, however, due to its complexity, we only use it for the computation of the initial irreducible graph. Following the lead of previous work [3, 33, 52], in the remainder of this section we will refer to an (irreducible) graph $K$ as better than an (irreducible) graph $K'$ if it has fewer vertices. However, our algorithm can be adapted to other quality criteria by minor changes.

In the preceding non-increasing reduction algorithm, the cyclic blow-up algorithm ties in at the point where an irreducible graph has been obtained. The basic idea at this point is to execute increasing structions, which initially increase the size of the current graph again. The hope here is that the structure of the current graph changes in such a way that individual subgraphs become reducible again for the non-increasing reduction algorithm of the previous section. Ideally, we are therefore able to obtain a smaller current graph than before.

In general the cyclic blow up algorithm can be described as follows (see Algorithm 3): Similar to local searches we manage two graphs $K$ and $K^\star$ during the process. $K^\star$ is the best graph found so far, i.e. the graph with the least number of vertices and $K$ is the current graph, which we try to reduce to get a better graph $K^\star$. Both graphs are initialized with the graph obtained by the non-increasing reduction algorithm of the previous section to the input graph (lines 3,4). The algorithm then runs in iterations consisting of two phases, a *blow-up phase* and a *reduction phase*. During the blow-up phase a set of increasing structions is applied to the current graph, resulting in a new graph $K'$ (line 8). More details about this phase are given in Section 6.2.1. This graph $K'$ is then reduced using the non-increasing reduction algorithm, resulting in a reduced graph $K''$ (line 12). We then

**Algorithm 4** Blow-Up Phase

```
1: input graph G
2: function BlowUp(G)
3:     while stopping criterion not reached do
4:         v ← PickVertex(G)
5:         G' ← Struction(G, v, n_max)
6:         if G' = G then
7:             return G
8:         G ← G'
9:     return G
```

decide whether the new graph $K''$ or the old one $K$ should be used as the current graph for the next iteration. Intuitively it can be advantageous to accept a graph $K''$ even if it has more vertices than $K$ to avoid local minima. The decision between $K''$ and $K$ is handled by the Accept function (line 14), for which two strategies are shown in Section 6.2.2. If the graph $K''$ is rejected for the next iteration, it would be created again in the next iteration if the blow-up and reduction phase are executed in the same way. In section 6.2.3 simple approaches are presented to avoid such recurring circles.

Since we generally cannot always obtain an empty graph, we need a stopping criterion to abort the algorithm prematurely (line 6). In Section 6.2.4 we introduce the stopping criterion used and also show current weaknesses of it.

Finally we want to note, that we use the full set of reduction rules of the non-increasing reduction algorithm for the calculation of an initial reduced graph, while we omit the critical set reduction in subsequent Reduce() calls.

### 6.2.1. Blow-Up Phase

The starting point of this phase is that the current graph $K$ is irreducible for the non-increasing reduction algorithm, i.e. in particular no more decreasing or plateau structions can be applied. The goal of this phase is to change the structure of the current graph by application of increasing structions in such a way that it can potentially be reduced in the subsequent reduction phase.

The general procedure of this phase is as follows (see Algorithm 4): We first select a vertex from a candidate set $\mathcal{C}$ on which we want to apply the struction (line 4). This set $\mathcal{C}$ consists of all vertices in the current graph $K$ which have not been explicitly excluded for selection during the algorithm. This vertex selection is a crucial part of the whole algorithm, because depending on the selection the struction can create a large number of new vertices and the size of the transformed graph can increase drastically. If the transformed graph cannot be reduced afterwards, or only slightly, we would stray away from our main goal of creating a small, easy to solve irreducible graph. We therefore present a variety of different vertex selection strategies in Section 6.2.1.1.

Afterwards, the struction is applied to the selected vertex $v$. In order to avoid the complete execution of a struction, which would be extremely costly in terms of running time by creating a large number of new vertices including adjacencies, a maximum number

of new vertices $n_{max}$ is specified. The execution is aborted as soon as the number of new vertices exceeds $n_{max}$. In this case the vertex $v$ is excluded from the candidate set $\mathcal{C}$ and the blow-up phase is terminated (line 5). The vertex $v$ will become selectable again as soon as the corresponding struction would create another transformed graph, i.e. its neighborhood $N(v)$ changed.

In general, this process is repeated until a certain stopping criterion is reached (line 3). In Section 6.2.1.2 we present two different criteria for this purpose.

#### 6.2.1.1. Vertex Selection Strategies

Basically, the goal of the vertex selection procedure is to find an increasing struction that calculates a transformed graph from the current one, which can then be reduced to an overall simpler graph than the original one. In general, however, it is very difficult to estimate in advance to what extent the transformed graph can be reduced without actually performing the reduction phase. Most of the strategies presented therefore aim at increasing the graph size by only a few vertices. This approach seems promising, since the graph only needs to be reduced by a few vertices to result in an overall vertex reduction. In addition, this also gives us an advantage in terms of running time, since the execution of the corresponding struction takes less time. The value by which the number of vertices changes by applying a struction to a vertex $v$ depends on the number of newly created vertices. This in turn is determined by the number of independent sets in the neighborhood of $v$ having greater weight than $v$. In general, determining the number of independent sets is $\mathcal{NP}$-complete and can reach a maximum of $2^n$ sets for graphs of size $n$ with empty edge sets [48]. Therefore, in addition to an exact selection strategy that always selects the vertex with the minimum vertex increase, we also present two heuristics that attempt to find vertices with comparable increases.

The presented strategies all make use of an addressable priority queue to manage the candidate set $\mathcal{C}$ (for more details about this data structure we refer the reader to [42]). They differ in the used KEY function, which describes a local selection criterion, i.e. it only takes the neighborhood of a vertex into account. The vertex selection is done by the DELETEMIN function, which removes and returns an element with minimal key from the queue. As soon as a change in the neighborhood of a vertex occurs, i.e. a new vertex is added, removed or the weight of a neighbor changes, the key of the corresponding vertex is updated using the provided CHANGEKEY function. In the following, the different vertex selection strategies are presented, highlighting their advantages and disadvantages.

**Random Selection.** This is the simplest strategy, which selects a vertex uniformly at random. However, experiments have shown that the random choice of vertices results in the application of structions that significantly increase the graph size. As a result, the algorithm either terminates after applying a few transformations or a lot of time must be spent to reduce the graph to its original size again. For this reason there is a need for more sophisticated selection strategies.

**Degree Selection.** This strategy selects a vertex with a minimum degree, thus minimizing the worst-case number of newly created vertices. We can calculate the key in constant

time, but the worst-case number is not necessarily meaningful, depending on the graph structure. vertices with a high degree would either not be selected at all or selected very late with this strategy, but can still be a good choice: If a vertex $v$ has a dense neighborhood, the number of independent sets in $N(v)$ can be much less than for a vertex $v'$ of lower degree with sparse neighborhood $N(v')$. Depending on the weight distributions between center vertex $v$ and its neighborhood $N(v)$, the number of new vertices can also be much less than the number of independent sets; The greater the weight of $v$ compared to its neighborhood $N(v)$, the fewer independent sets have greater weight than $v$.

**Vertex Increase Selection.** This strategy selects the vertex whose corresponding struction increases the number of vertices of the graph the least. In order to calculate the key for a vertex $v$, an exhaustive search is used to find the number $x$ of independent sets with a weight greater than $v$. Since in the blow-up phase only structions are executed which create less than $n_{max}$ vertices, we stop the search as soon as the number of sets found reaches $n_{max}$. The key is thus the difference between $x$ and $\delta(v) + 1$, where the latter is the number of removed vertices. A weakness of this selection strategy is that it can be very expensive in terms of running time, Especially for vertices with large and sparse neighborhoods the key calculation can become very time consuming.

**Approximate Vertex Increase Selection.** This strategy works similarly to the vertex increase selection. However, instead of calculating all independent sets in the neighborhood of $v$, only independent sets up to a size of two are considered. This results in a lower bound $L$ for the number of independent sets [47]. Two vertices form an independent set, if they are not connected by an edge, so the number of such sets can be calculated in $\mathcal{O}(\Delta^2)$ time, where $\Delta$ is the maximum degree of $G$. Analogous to the vertex increase selection key for a vertex is calculated by the difference between $L$ and $\delta(v) + 1$.

Since the lower bound $L$ can be far below the actual number of newly created vertices, we use a *tightness-check*: It is passed if less than $L' = \lceil \beta \cdot L \rceil$ new vertices with $\beta \in (1, \infty)$ are created by the corresponding struction.

In principle, the approximate vertex increase selection then works as follows: We select a vertex $v$ from the queue with minimal key and perform the tightness-check. If it fails, we know that at least $L'$ new vertices are created by the corresponding struction. Therefore $L'$ forms a tighter bound for the number of new vertices, and we reinsert $v$ to the queue again using the bound $L'$. We repeat this process until we find a vertex that passes the check and return it as the result of the vertex selection.

In practice, we combine tightness-check and struction execution by executing the corresponding struction with a maximum number $L'$ of new vertices.

#### 6.2.1.2. Stopping Criteria

Generally, we stop the blow-up phase as soon as our candidate set $\mathcal{C}$ is empty. This is the case when we have excluded all remaining vertices in the current graph $K$ from $\mathcal{C}$ or $K$ is empty. In addition, we always use another criterion to avoid excessive increases in graph size over time. For this purpose we present two criteria, which we will compare to each other in the later evaluation.

The first criterion allows a maximum number of structions $Z$ to be executed per phase. Since the growth of the graph heavily depends on the selected structions, this criterion is not adaptive.

The second criterion therefore terminates the blow-up phase if the current graph exceeds a percentage limit. Therefore the size of the graph at the beginning of the blow-up phase is recorded. We stop executing further structions as soon as the size of the current graph $K'$ exceeds the initial graph size by a factor of $\gamma \in [1, \infty)$.

### 6.2.2. Accept Strategies

After we have calculated a new graph $K''$ during an iteration of the cyclic blow up algorithm, we have to decide if we want to keep or abandon it. We present two simple Accept strategies, which are later evaluated in chapter 7.

Both procedures calculate the new best graph $K^\star$ from the minimum of $K^\star$ and $K''$. The first strategy accepts any calculated graph $K''$ for the next iteration. Since this also allows to keep graphs $K''$ which are much larger than the current graph $K$, we investigate a second strategy. This accepts $K''$ only if it has fewer vertices than the current graph, i.e. $\text{Vertices}(K'') < \text{Vertices}(K)$. Other strategies could use a midway between the two extremes by accepting a graph $K''$ whose size is at most a factor $\eta \geq 1$ worse than $K$. Our two procedures are derived from this by $\eta = \infty$ or $\eta = 1$.

### 6.2.3. Cycle Avoidance Strategies

In this section we describe two simple approaches that should help to avoid that a rejected graph $K''$ is produced again in the next iteration by running the same blow-up and reduction phase.

**Randomized Tie Breaking.** This approach introduces a random component into the blow-up phase. For this purpose, ties between vertices, i.e. vertices with the same keys, are randomly broken during vertex selection. We achieve this by using floating point keys whose decimal part corresponds to random noise. The original key can be reconstructed by cutting off the decimal part.

**Tabu Mechanic.** This strategy avoids the execution of the same blow-up phase by excluding vertices from the candidate set $\mathcal{C}$ for vertex selection. After a calculated graph $K''$ has been rejected, all vertices from the previous blow-up phase become unselectable. These vertices only become available again after their neighborhood has changed, which causes the corresponding struction to generate a different transformed graph.

### 6.2.4. Stopping Criteria

If we have obtained an empty graph during the algorithm, we essentially solved the problem. Otherwise, we sooner or later have to make the decision to output the current best graph as the result. As termination criteria for the algorithm we consider two factors. First we want to avoid that the size of the current graph $K$ distances too much from that

of the best graph $K^\star$. Therefore we abort the algorithm as soon as the size of the current graph $K$ exceeds the size of the best graph $K^\star$ by a factor $\alpha \in [1, \infty)$. Reaching this criterion depends on the used Accept strategy. Therefore we also count the number of unsuccessful iterations, i.e. iterations in which the new graph $K''$ has been rejected. Our second criterion aborts the algorithm if this value exceeds some constant $X \in [1, \infty)$.

At this point we would like to state that this is a very simple stopping criterion. In general, the size of the best graph in the algorithm can decrease very slowly or even exhibit oscillatory behavior. However, our current stopping criterion does not prevent this in general. This can cause the algorithm to take a long time to improve the current graph or even fails to improve it at all. In practice, it might be more practical to stop the algorithm at this point and try to calculate an optimal solution on the best graph $K^\star$. This could result in a reduction of the overall runtime for calculating an optimal solution. Therefore, a more general stopping criterion would be desirable, which aborts the algorithm in such cases. A stopping criterion similar to the one used by Hespe et al. [31], which measures the rate of change of the current graph and aborts when it becomes too small, is not directly applicable here. The problem is that we generally want to allow graph size increases during a blow-up phase. The question of a suitable stopping criterion is therefore still open at this point.

# 7. Evaluation

## 7.1. Experimental Setup

### 7.1.1. Environment

We ran all the experiments on a machine with four Octa-Core Intel Xeon E5-4640 processors running at 2.4 GHz, 512 GB of main memory, 420 MB L3-Cache and 48256 KB L2-Cache. The machine runs Ubuntu 18.04.4 and Linux kernel version 4.15.0-96. All algorithms were implemented in C++11 and compiled with g++ version 7.5.0 with optimization flag -O3. All algorithms were executed sequentially with a time limit of 1 000 seconds. The experiments for heuristic methods were performed with five different random seeds.

### 7.1.2. Datasets

For our experimental evaluation we use data sets already known from previous works on the maximum (weight) independent set problem [5, 12, 33]. Before we introduce the instance types in detail, we first want to note that all instances except the OSM instance family are unweighted. To be suitable for maximum weight independent sets, a commonly used approach is to assign random weights to the vertices. Following the example of previous work [12, 33, 37], we use uniformly distributed weights in the interval $[1, 200]$. Basic properties like number of vertices $|V|$ and edges $|E|$ of our test instances can be found in the appendix in Tables A.1-A.4.

**Map Labeling (OSM).** These instances are label conflict graphs obtained from OpenStreetMap(OSM) data [45] using a method described by Barth et al. [9] to generate map labelings. In general, label conflict graphs can be obtained by considering points of interest (POIs) of a region, that should be labeled in a map. For each label we therefore create a vertex and connect it to other vertices, if their labels overlap each other. The weights are chosen proportionally to the importance of the label respectively, so that by obtaining an MWIS on this graph we determine an overlap-free labeling that maximizes the sum of importance of labels. More specifically, our instances were generated by considering a dynamic setup, where vertices are associated with labels and certain activity intervals, corresponding to the time they are displayed [9]. Thereby we consider the same instances as Cai et al. [12], which were created for three different Activity Modes (AM1,AM2,AM3) for states in North America. Following the lead of previous work, we omit instances with less than 1 000 vertices, since they are easy to solve [12, 33].

**Stanford Large Network Dataset Repository (SNAP) [34].** This data set contains large scale network instances from many different areas, namely collaboration networks, communication networks, road networks, social networks, peer-to-peer networks and web crawl networks. These instances are unweighted and are a well-known benchmark data set for the maximum independent set problem [3, 14, 17].

**Mesh.** This instance family consists of the dual graphs of well-known triangle meshes. During the conversion, vertices of degree zero and one have already been iteratively removed from the graph. Originally this instance family was motivated by an application of Sanders et al. [50]: To efficiently process a triangulation in hardware, a small subset of triangles covering all edges of the mesh is required. Since adjacent facets in the mesh are adjacent vertices in the dual graph of the mesh, this problem corresponds to finding a vertex cover in the dual graph and thus finding an independent set.

**Finite Elements (FE).** These instances were obtained from 3d meshes, which stem from simulation using the finite element method. Originally these graphs were used as benchmark instances for graph partitioning algorithms[23, 60].

Furthermore, we have also tested our reduction rules on benchmark instances for the maximum weight clique problem [53]. To transfer these instances to the maximum weight independent set problem, we need to calculate the complement graph which is feasible since these instances only consist of a few hundred to thousand vertices. However, we have observed that this results in very dense graphs, which are already (almost) irreducible for our reduction algorithms. This behavior has already been observed by Akiba and Iwata [3] on DIMACS graphs of the maximum clique problem for different reduction rules as well. In the following, we will therefore focus on the benchmark instances for the maximum (weight) independent set problem.

### 7.1.3. Methodology

During the evaluation we present some of our collected data by using three types of plots, which we explain briefly.

**Cactus Plots.** These plots show the number of instances solved over time for exact solution methods. For each algorithm and instance, the time needed to solve it (if possible within the time limit) is logged. At these times we get one step for the line of the corresponding algorithm, in which the solved instances increase by one.

**Convergence Plots.** For a single test instance, convergence plots show how the solution quality of an optimization algorithm changes over time. For reduction algorithms we plot the size of the smallest irreducible graph found, while for optimization algorithms for the maximum weight independent set problem we consider the weight of the largest weight independent set found. We therefore obtain a new tuple (time,value) consisting of the

current best solution (value) and current time for each algorithm, as soon as it has found a new best solution.

An accumulated view for multiple seeds or execution runs can be obtained as described by Sanders and Schulz [51] by event-based geometric means: For this purpose, we log triples (time,value,seed) during the measurement, which include besides the time and solution quality (value) also the used seed. Afterwards we create a list $S$, containing all measured triples sorted ascending by time. For each seed we manage a current value, which we initialize by the first measured triple of the corresponding seed. Then we iterate over $S$ and update the current value for the seed of the current triple (time,value,seed). From the current values of all seeds we determine the geometric mean $G$ and append a tuple (time,$G$) to our (initially empty) result list $S_g$. To additionally accumulate over multiple instances, we can use the same procedure by simply logging instance names instead of seeds.

Note that we always set the current values for determining the geometric means to a minimum value of one, even if we have obtained an empty graph. This way we avoid that the geometric mean drops to zero as soon as one of the current values has reached zero.

**Performance Plots [18].** A performance plot consists of a performance profile in form of a curve for each evaluated optimization algorithm. Thereby performance profiles are distribution functions for a specific performance metric that allow to benchmark and compare optimization algorithms with each other. In our case, for reduction algorithms we use the size of the obtained irreducible graph as metric, while for exact solvers we consider the total time needed to both calculate an optimal solution and prove its optimality. Furthermore, we refer to the output of an algorithm as the irreducible graph size or total run time respectively. In particular, a performance profile of an algorithm maps a variable $\tau \geq 1$ to the fraction of instances on which its output is not worse than the best output found by any of the evaluated algorithms multiplied by $\tau$. For instance for $\tau = 1$ we obtain the fraction of all instances on which the algorithm calculates the best output, whereas for $\tau = 2$ we get the fraction of all instances on which the algorithm produces an output that is at most twice as bad as the best found output on this instance.

### 7.1.4. Experimental Design

In the following section we perform a parameter tuning to find reasonable configurations for the non-increasing and cyclic blow-up algorithm. This results in three concrete reduction algorithms, namely one configuration for the non-increasing algorithm and two for the cyclic blow-up algorithm. While the first cyclic blow up configuration $C_{\text{strong}}$ calculates the smaller reduced graphs, $C_{\text{fast}}$ takes less time to calculate larger reduced graphs. We then perform a comparison with other state-of-the-art solvers for the maximum weight independent set problem including both exact and heuristic methods. For this purpose we equip the branch-and-reduce framework of Lamm et al. [33] with our three reduction algorithms. When we refer to this framework in the following, we will omit the addition of the authors, since we do not use any alternative branch-and-reduce framework.

| Graph | NonIncreasing [no struction] | | Plain orig. struction | | Plain mod. struction | | Plain ext. struction | | Plain ext. red. struction | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *n* | *t* | *n* | *t* | *n* | *t* | *n* | *t* | *n* | *t* |
| fe_body | 15992 | 0.88 | 44430 | 0.02 | 44469 | 0.02 | **1437** | 0.22 | 1753 | 0.32 |
| fe_sphere | 15269 | 0.41 | 16386 | 0.01 | 16386 | 0.01 | **5416** | 0.50 | 5448 | 0.29 |
| buddha | 107265 | 14.53 | 1087716 | 0.18 | 1087716 | 0.18 | **387** | 2.11 | 2804 | 2.04 |
| ecat | 26270 | 9.33 | 684496 | 0.22 | 684496 | 0.35 | **1995** | 3.01 | 9514 | 2.47 |
| georgia-AM3 | **861** | 3.13 | 1462 | 0.01 | 1458 | 0.00 | 870 | 0.29 | 870 | 0.65 |
| rhode-island-AM2 | **1103** | 0.51 | 2653 | 0.03 | 2653 | 0.02 | 1801 | 2.64 | 1798 | 4.17 |
| roadNet-PA | 35442 | 3.83 | 971421 | 0.52 | 991000 | 0.44 | **771** | 1.51 | 3611 | 1.56 |
| soc-LiveJournal1 | 29419 | 98.37 | 4266942 | 7.69 | 4283809 | 7.78 | **11947** | 104.27 | 12210 | 145.88 |
| web-NotreDame | 6052 | 1.49 | 299118 | 0.19 | 299724 | 0.20 | 2109 | 0.61 | **2061** | 0.58 |

Table 7.1.: Obtained irreducible graph size by basic reduction rule set (NonIncreasing [no struction]) and each struction variant and time (in seconds) required to compute it. The global best irreducible graph size is highlighted in bold.

## 7.2. Parameter Tuning

We notice, that some parameters, such as the struction variant used in the non-increasing reduction algorithm, can be determined almost independently of the choice of other parameters. Other highly interdependent parameters such as the maximum number of unsuccessful blow-ups $X$, the maximum number of vertices $n_{max}$ allowed to be created during a struction application and the maximum struction degree $d_{max}$ had to be evaluated simultaneously using a grid search. However, with a large number of parameters, this would lead to an unreasonable runtime effort, so we will limit ourselves to looking at just a few configurations to find a good one. For the quality of an irreducible graph found, on one hand we take its size, i.e. the number of vertices, into account. If one configuration finds a smaller irreducible graph in a slower time than another, they are non-comparable by this criterion. Therefore, on the other hand, we also use the total runtime needed to solve an instance as metric, i.e. the time needed to calculate the irreducible graph and subsequent branch-and-reduce runtime.

### 7.2.1. Non-Increasing Reduction Algorithm

First, we evaluate the different weighted struction variants in order to find a suitable selection for the subsequent experiments. We therefore form an evaluation set that consists of nine instances from our data sets. From each of the instance groups OSM, finite elements and mesh, we select two instances, while we take three SNAP instances of different network types. These instances are chosen in a way that the obtained irreducible graphs by the reduction rules of Lamm et al. [33] consists of several hundred to thousands of vertices, so that there is room for improvement. In the following we refer to these reduction rules by Lamm et al. as *basic reduction rule set* or simply *basic reductions*. In the following tables, we use the notation NonIncreasing[X] for the non-increasing reduction algorithm from Section 6.1 where X is a particular struction variant or takes the value "no struction" if we use the non-increasing reduction algorithm without any struction application. Furthermore we write Plain X, if we reduce a graph by using only using a single struction variant X.

| Graph | NonIncreasing [no struction] | | NonIncreasing [orig. struction] | | NonIncreasing [mod. struction] | | NonIncreasing [ext. struction] | | NonIncreasing [ext. red. struction] | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *n* | *t* | *n* | *t* | *n* | *t* | *n* | *t* | *n* | *t* |
| fe_body | 15992 | 0.88 | 14018 | 0.86 | 15333 | 1.51 | **1167** | 0.82 | 1206 | 0.82 |
| fe_sphere | 15269 | 0.41 | 15269 | 0.41 | 15269 | 0.45 | **3540** | 1.26 | 3578 | 1.38 |
| buddha | 107265 | 14.53 | 71574 | 13.96 | 227833 | 22.10 | **103** | 8.89 | 263 | 8.73 |
| ecat | 26270 | 9.33 | 17626 | 9.56 | 121122 | 17.14 | **251** | 7.55 | 919 | 7.83 |
| georgia-AM3 | 861 | 3.13 | 861 | 3.11 | 860 | 2.97 | **780** | 3.33 | 781 | 3.40 |
| rhode-island-AM2 | 1103 | 0.51 | 1100 | 0.64 | 1098 | 0.53 | **853** | 2.49 | 901 | 1.96 |
| roadNet-PA | 35442 | 3.83 | 20572 | 4.10 | 63621 | 6.93 | **282** | 3.16 | 377 | 3.31 |
| soc-LiveJournal1 | 29419 | 98.37 | 23107 | 127.25 | 42270 | 118.01 | 4319 | 153.87 | **4293** | 147.40 |
| web-NotreDame | 6052 | 1.49 | 5511 | 1.54 | 6348 | 1.39 | **516** | 1.38 | 544 | 1.41 |

Table 7.2.: Obtained irreducible graph sizes by non-increasing reduction algorithm for each struction variant and time (in seconds) required to compute it. The global best irreducible graph size is highlighted in bold.

#### 7.2.1.1. Struction Variant

In the first experiment we execute the different struction variants on the evaluation set to get an impression of their practical potential. In detail, we use an incremental reduction rule application, where we execute the corresponding struction first as a decreasing reduction and then as a plateau reduction without using any further reduction rules. We compare both the size of the irreducible graph and the run time required to obtain it with the basic reduction rule set. In particular, we use the non-increasing reduction algorithm from Section 6.1 without any struction variant. This corresponds to the basic reduction algorithm (see Section 4.1) with the only difference that we apply the generalized fold reduction after the weighted critical set reduction. The results can be found in Table 7.1. We find that both the original and the modified weighted struction have shorter run times but at the same time obtain much larger irreducible graphs than the other methods. In detail, the geometric mean of the runtime of the latter two is about 20 times as much as for the other two and even have run times on the two OSM instances `georgia-AM3` and `rhode-island-AM2` that are more than two orders of magnitude greater. At the same time, the graph sizes of the extended and extended reduced weighted struction are about 35 times smaller on geometric mean, and especially on all SNAP instances and the mesh instance `buddha` more than two orders of magnitude smaller. This is primarily due to the weight limitation for the center vertex, which means that generally only few struction applications are available for these types. For the extended and extended reduced struction, we can see that the irreducible graphs of the OSM instances are larger than those obtained by the basic reductions. While on the `georgia-AM3` instance, this difference only amounts to nine vertices, the reduced graphs on the `rhode-island-AM2` instance are larger by a factor of 1.6. However, we can achieve significant improvements of several orders of magnitude on the remaining instances. This can be explained by the inclusions of existing reduction rules and new reducible subgraphs found in Section 5.3. Except for the `rhode-island-AM2` and `soc-LiveJournal1` instances, the run times of both struction variants are also lower than for the existing reduction rule set. On the `georgia-AM3` instance this difference even goes up to a factor of ten for the extended weighted struction.

In the second experiment we execute the non-increasing reduction algorithm with the four different struction variants and compared them to the non-increasing reduction

| Graph | extended | | $\text{extended}_{\text{g}}$ | | $\text{extended}_{\text{gn}}$ | | $\text{extended}_{\text{gnc}}$ | |
|---|---|---|---|---|---|---|---|---|
| | $n$ | $t$ | $n$ | $t$ | $n$ | $t$ | $n$ | $t$ |
| fe_body | 1167 | 0.82 | 1177 | 0.20 | 1162 | **0.17** | 1162 | 0.19 |
| fe_sphere | 3540 | 1.26 | 3662 | 0.33 | 2961 | **0.31** | 3510 | 0.30 |
| buddha | 103 | 8.89 | 86 | 2.12 | 86 | **1.90** | 86 | 2.07 |
| ecat | 251 | 7.55 | 258 | 2.66 | 274 | **2.29** | 253 | 2.92 |
| georgia-AM3 | 780 | 3.33 | 796 | 0.33 | 796 | **0.29** | 796 | 0.29 |
| rhode-island-AM2 | 853 | 2.49 | 845 | 1.27 | 845 | **1.20** | 835 | 1.46 |
| roadNet-PA | 282 | 3.16 | 282 | 1.11 | 300 | **1.07** | 302 | 1.27 |
| soc-LiveJournal1 | 4319 | 153.87 | 4319 | **22.33** | 4319 | 22.72 | 4293 | 31.45 |
| web-NotreDame | 516 | 1.38 | 516 | 0.50 | 516 | **0.35** | 516 | 0.38 |

Table 7.3.: Obtained irreducible graph size by non-increasing reduction algorithm with extended reduced weighted struction but without **g**eneralized fold/**n**eighborhood clique removal/**c**lique reduction and time (in seconds) required to compute it. The global best time is highlighted in bold.

algorithm without any struction variant. Again, we measure the irreducible graph sizes as well as the time required to obtain them; the results can be found in Table 7.2. We get a similar pattern as in the previous experiment: The first two struction variants produce much larger irreducible graphs than the last two, in the geometric mean again by one order of magnitude more. The biggest difference can be found between the modified and the extended weighted struction on the `buddha` instance, where the graph sizes differ by a factor of more than 2 200. Except for the modified weighted struction, we can always obtain smaller irreducible graphs compared to using no struction. We also observe that the discrepancy of the obtained graph sizes between extended and extended reduced weighted struction decreases, which we explain by the fact that the struction is applied less often in both algorithms, since many structures are already reduced by other reduction rules. Nevertheless, the non-increasing reduction algorithm using the extended weighted struction produces slightly smaller irreducible graphs. Therefore, the two experiments lead us to the conclusion that we will use the non-increasing reduction algorithm with the extended weighted reduced struction.

However, in direct comparison, we find that the overall run times between the exclusive use of struction in the first experiment and the integrated form in the non-increasing algorithm in the second experiment are higher. Overall, we believe that this behavior can be explained by the fact that we get a larger runtime overhead due to an increased number of (unsuccessful) applicability checks. In the following section, we therefore try to find a subset of reduction rules with which we can obtain comparable irreducible graph sizes in a shorter runtime.

#### 7.2.1.2. Reduction Rules

We have already noticed that the runtime increases on many instances as soon as we use the extended weighted struction in the non-increasing algorithm instead of in a standalone mode. In this section, we therefore want to examine to what extent the overall runtime

can be reduced if other expensive reduction rules, which are already included by the extended weighted struction, are not used. This is of particular importance since we later execute the non-increasing algorithm as subroutine in the cyclic blow up multiple times. Therefore we execute the non-increasing algorithm again on the instances with the extended weighted struction and successively disable the application of the generalized fold, the clique neighborhood reduction and the clique reduction. The results can be found in Table 7.3. We first note that no significant changes occur in the obtained irreducible graph sizes. However, especially without using the generalized fold, but also without the application of the clique neighborhood reduction, a large run-time advantage can be obtained. In particular we can always obtain the best run times that are faster by a factor of four with the exception of the `soc-LiveJournal1` instance, where we get a negligibly higher runtime of less than 2%. We assume that the strong negative influence of the generalized fold reduction is due to the fact that several recursive calls of the branch-and-reduce algorithm (using a small subset of reduction rules that are favorable in terms of runtime) are executed on induced neighborhood graphs to determine the applicability. Finally, this experiment leads us to execute the non-increasing reduction algorithm without the usage of the generalized fold and clique neighborhood reduction in the following sections.

### 7.2.2. Cyclic Blow-Up Algorithm

In the following sections we want to find good parameter configurations for the cyclic blow-up algorithm. Based on the acceptance and cycle avoidance strategy used, we obtain three basic configurations for the cyclic blow-up algorithm: If we always accept the new graph after a blow-up phase, we call the configuration $C_{accept}$, otherwise we refer to these configurations as $C_{tabu}$ and $C_{tie\ break}$, dependent on the cycle avoidance strategy used. For these three configurations, we determine reasonable values for the remaining parameters in the following experiments. Finally, by selecting one of these basic configurations with two different parameter sets we obtain two configurations of the cyclic blow-up algorithm, which we compare with state-of-the-art solvers in the subsequent section.

#### 7.2.2.1. Vertex Selection Strategy

The goal of this section is to select a vertex selection strategy for the following experiments. For this purpose we run the cyclic blow up algorithm with the different vertex selection strategies and a single struction application per blow up phase. For the approximate vertex increase selection we used $\beta = 2$ as tightness check factor. By using accumulated convergence plots of the graph size for different instance families we can derive a statement about quality and convergence independent of the stopping criterion. Therefore, we disable the stopping criterion of the cyclic blow up algorithm and do not use any further restrictions like maximum struction degree or maximum number of new vertices per struction. At this point, we only consider instances on which the non-increasing reduction algorithm generates irreducible graphs with more than 100 vertices. The measurement of the graph size starts immediately before we execute the first blow up phase, while we start the timing at the beginning of the cyclic blow up algorithm. The first logged point of

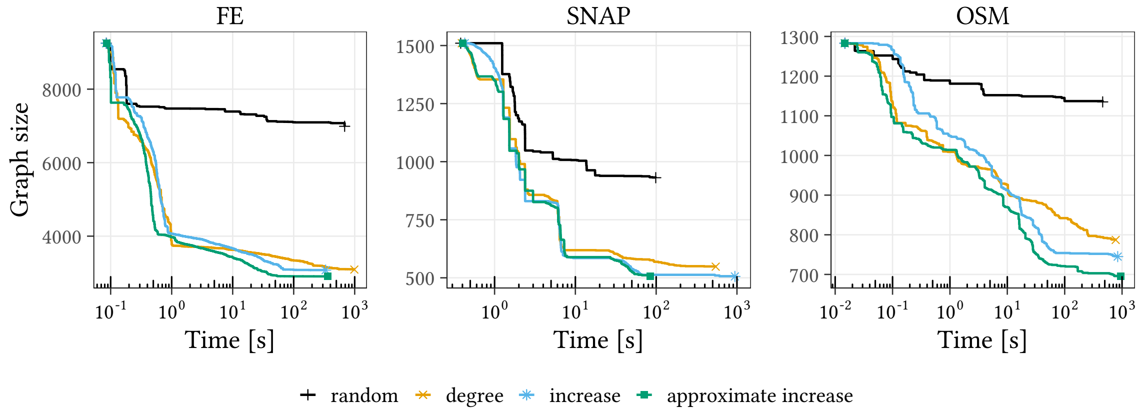



Figure 7.1.: Accumulated convergence plots for configuration $C_{\text{tabu}}$ with different vertex selection strategies.

each vertex selection strategy thus corresponds to the size of the graph calculated by the non-increasing algorithm and the time required to obtain it.

The results for configuration $C_{\text{tabu}}$ are shown in Figure 7.1, we obtain similar results for the other two configurations that can be found in the appendix in Figures A.1-A.3. Since already six of the ten mesh instances can be reduced to an empty graph by the non-increasing algorithm and the remaining four reduced graphs consist of a few hundred vertices, we omit them at this point. First of all, we see that we are always able to obtain a smaller irreducible graph than with the non-increasing reduction algorithm, using any of the strategies. However, as expected, we also find that the random selection performs worst. It produces both the largest irreducible graphs and takes the most time to calculate them. In particular, on the FE family the smallest obtained graphs by the random selection are with a factor of 2.4 more than twice as large as those of the best strategy (approximate increase vertex selection), on the SNAP and OSM family we get a factor of 1.8 and 1.6. By selecting a vertex with a minimum degree, we can further reduce both convergence and graph size. Finally, we see that using the increase and approximate increase selection we can always get the smallest graphs, and in the case of SNAP instances these are even a factor of three smaller than with the non-increasing reduction algorithm. However, we also see that the increase selection needs more time than the degree selection to reduce the graph size, especially at the beginning of the algorithm. By using the approximate increase selection we overcome this problem and can obtain the smallest graph sizes. On the FE family, the graph sizes are by a factor of 3.2 smaller than the ones obtained by the non-increasing algorithm, on the SNAP and OSM families we obtain the factors 3 and 1.8. Since we obtain similar results for the other two configurations $C_{\text{accept}}$ and $C_{\text{tie break}}$, in the following experiments we use the approximate increase vertex selection for all strategies.

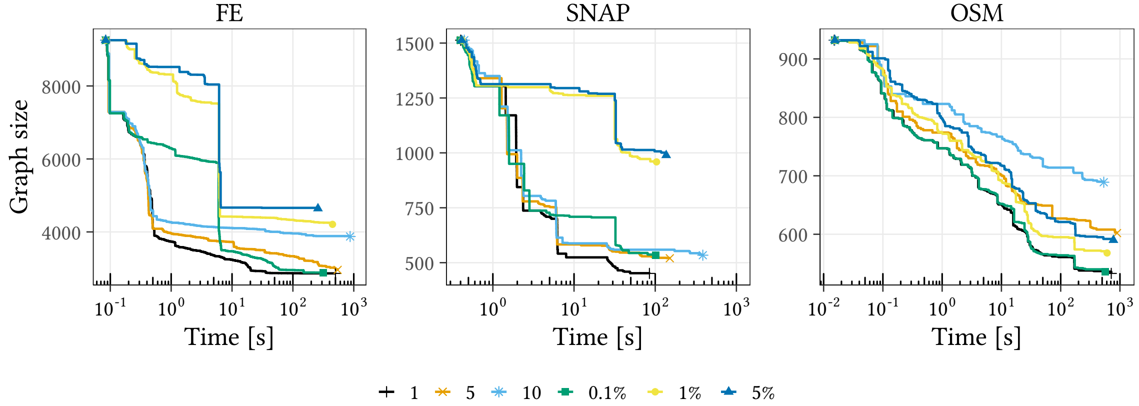



Figure 7.2.: Accumulated convergence plots for the cyclic blow up algorithm with configuration $C_{\text{tabu}}$ and different blow-ups per phase limit.

#### 7.2.2.2. Struction Application Limit Per Blow-Up Phase

In this section we want to determine proper limits for the amount of struction applications per phase for each of the three basic configurations. For this purpose, we run the cyclic blow-up algorithm for each configuration with different struction application limits and again we looked at the convergence of the graph sizes. We use both fixed struction limits, which only allow a certain number of struction applications per phase, and percentage limits, where the current graph size may only increase by a defined fraction compared to the graph size at the beginning of the blow-up phase. For the fixed struction limits we use a maximum of one, five and ten struction applications per phase, while for the percentage limits we examine values of 0.1%, 1% and 5%. As an example, Figure 7.2 shows the convergence plots for configuration $C_{\text{tabu}}$. Convergence plots for all configurations can be found in the appendix in figures A.4-A.6. For the configurations $C_{\text{accept}}$ and $C_{\text{tabu}}$, we found that the quality decreases with an increasing number of struction application per blow-up phase, so we get the best results by one struction per phase. For configuration $C_{\text{tie break}}$, however, we get the best results for five struction applications per phase. A closer look at $C_{\text{accept}}$ and $C_{\text{tabu}}$ shows that for both percentage and fixed limits, as the number of structions per phase increases, both the convergence speeds decrease and the minimum reduced graph sizes increase. Especially on the SNAP instances, we get about twice as large reduced graph sizes in the end for both configurations with a 1% and 5% phase limit than with a limit of a single struction application per phase. Furthermore, we can observe that multiple struction applications per phase tend to have a worse effect on graph sizes for $C_{\text{accept}}$ than for $C_{\text{tabu}}$. When using fixed struction limits with up to ten applications per phase we obtain with $C_{\text{tabu}}$ on all instance families a maximum of 15% larger graphs than with one application. With $C_{\text{accept}}$ this is more than twice as much with 35% on the FE family.

Looking at the configuration $C_{\text{tie break}}$ we see that we can achieve the best results in terms of convergence speed and reduced graph size with a maximum of five or ten struction

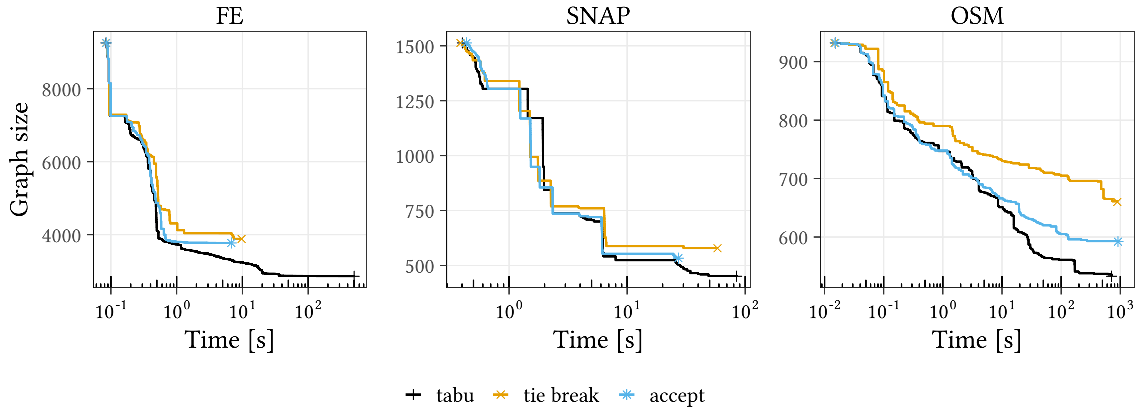



Figure 7.3.: Accumulated convergence plots for the three basic configurations with best found blow-ups per phase limit.

applications per phase, while five applications are slightly superior. Here, too, we get the relatively largest deviations in the minimum reduced graph sizes on the SNAP instances. While ten struction applications provide a graph that is only about 1% larger in the end than five applications, these fractions vary between 80% and 120% for all other variants.

#### 7.2.2.3. Base Configuration

In the following we compare the individual configurations $C_{\text{accept}}$, $C_{\text{tabu}}$ and $C_{\text{tie break}}$ with each other using the determined struction limits per phase from the previous section. Based on this, we want to determine a base configuration that we will use for the following experiments. Therefore, Figure 7.3 shows the convergence plots for the base configurations with the determined struction limits per phase. Here, we can see that we always have a comparable or better convergence for configuration $C_{\text{tabu}}$ in comparison to the other configurations and always get the smallest geometric mean of the reduced graph sizes. On closer examination we can see that all configurations first compute similar sized reduced graphs over time until the graphs of configuration $C_{\text{tie break}}$ become always larger than those of configurations $C_{\text{accept}}$ or $C_{\text{tabu}}$. Furthermore we see that $C_{\text{accept}}$ is always superior to configuration $C_{\text{tie break}}$ by geometric means of equal or smaller size of the reduced graphs. While $C_{\text{tie break}}$ cannot achieve significant changes on the FE instances after about one second, on SNAP instances this is the case after about six seconds. On the OSM family, $C_{\text{tie break}}$ can still achieve improvements in graph size until the time limit is reached, but they are still dominated by $C_{\text{accept}}$ and $C_{\text{tabu}}$. In total, the geometric mean of the reduced graph sizes on the FE instances of $C_{\text{tie break}}$ is about one third larger compared to $C_{\text{accept}}$, while they are about one quarter larger on the OSM and SNAP instances. If we look at the configuration $C_{\text{accept}}$, we see it calculates graphs with similar sizes compared to $C_{\text{tabu}}$ over a much longer time than $C_{\text{tie break}}$. However, $C_{\text{tabu}}$ can achieve larger improvements especially on the OSM instances towards the end and calculates on those about 10% smaller

| Graph | $C_{\text{fast}}$ | | | $C_{\text{strong}}$ | | | $C_{\text{full}}$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ |
| fe_sphere | 147 | 0.71 | 0.92 | 0 | 0.70 | **0.72** | 0 | 0.69 | 0.71 |
| alabama-AM3 | 456 | 1.51 | **4.03** | 0 | 32.27 | 32.31 | 0 | 33.81 | 33.85 |
| florida-AM3 | 661 | 0.44 | **2.85** | 267 | 42.65 | 45.46 | 267 | 1057.72 | - |
| georgia-AM3 | 587 | 0.47 | **10.49** | 425 | 12.63 | 31.83 | 425 | 1054.47 | - |
| mexico-AM3 | 483 | 0.39 | **1.75** | 0 | 20.83 | 21.10 | 0 | 22.62 | 22.89 |
| roadNet-PA | 0 | 1.03 | **1.27** | 0 | 1.42 | 1.92 | 0 | 1.07 | 1.33 |
| web-NotreDame | 2061 | 0.58 | **1.71** | 117 | 2.44 | 2.59 | 101 | 1056.04 | - |

Table 7.4.: Obtained irreducible graph sizes $n$, time $t_r$ (in seconds) needed to obtain them and total solving time $t_t$ (in seconds) for different configurations. The global best solving time $t_t$ is highlighted in bold.

reduced graphs. Therefore, in the following we will restrict ourselves on using the cyclic blow-up algorithm with the configuration $C_{\text{tabu}}$.

#### 7.2.2.4. Stopping Criterion And Maximal Struction Degree

In the last sections we have already found a configuration that allows us to compute small reduced graphs as well as to obtain them faster than with other configurations. Now we want to deal with the question to what extent it makes sense to use a larger graph instead of the smallest possible reduced graph in practice. The idea is to save time during the initial reduction step, which then becomes more available during the branch-and-reduce algorithm, resulting in a shorter overall solving runtime. For this purpose we equip the branch-reduce framework with our cyclic blow-up algorithm for computing an initial reduced graph from the input graph. All subsequent graph reductions before each branching step are performed by our non-increasing reduction algorithm, since we consider the additional time required for the cyclic blow-up algorithm to be excessively high.

Since we always discard a new reduced graph $K''$ after a blow-up phase if it has more vertices than our current best graph $K^\star$, the percentage factor $\alpha$ of the stopping criterion (see Section 6.2.4) has no impact. So at this point we just need to find a good value for the maximum number of unsuccessful blow-up phases $X$. The maximum number of vertices $n_{max}$ that can be created during a struction application serves as a further stopping criterion for our configuration: As soon as the number of new vertices exceeds the limit $n_{max}$, we stop the current blow-up phase. Since we only run a single struction application per blow-up phase, the graph $K'$ corresponds to the current graph $K$, so we terminate the cyclic blow-up algorithm. Finally, we examine the influence of the maximum vertex degree $d_{max}$ up to which a struction application is available.

Intuitively, we tend to get smaller reduced graphs the larger we choose our values for $X, n_{max}, d_{max}$, but this takes more time: While the first two values directly influence the runtime of the algorithm, a larger choice of $d_{max}$ gives us a larger reduction space. In the following, we consider different combinations of the three values, expressed as tuples $(X, n_{max}, d_{max})$. For each configuration, we run the branch-and-reduce algorithm on a small

set of test instances and log the total runtime $t_t$ to solve these instances as well as the initial reduced graph size $n$ and the time $t_r$ needed to obtain it. The results for three different configurations are listed in Table 7.4 with $C_{\text{full}} = (\infty, \infty, \infty)$, $C_{\text{strong}} = (512, 2048, 64)$ and $C_{\text{fast}} = (64, 512, 25)$.

The configuration $C_{\text{full}}$ executes the cyclic blow-up algorithm without any additional stopping criteria or maximum struction degree and always computes the smallest reduced graphs. However, this configuration is not able to obtain solutions on three out of seven instances, because the cyclic blow-up algorithm does not terminate within the time limit. In contrast, the configuration $C_{\text{strong}}$ always finds reduced graphs of the same size except for the SNAP instance `web-NotreDame`, but computes them faster and is always able to find an optimal solution on the seven instances. The last configuration $C_{\text{fast}}$ aims to achieve a good trade off between initial reduction and branch-and-reduce time. We notice, that we always compute larger or equally sized reduced graphs than by $C_{\text{strong}}$, for the `web-NotreDame` instance it is even larger than a factor of 17. However, except for the instance `fe_sphere` this configuration is able to find optimal solutions in less time than $C_{\text{strong}}$ due to the smaller reduction time. On the two OSM instances `florida-AM3` and `mexico-AM3` these times are up to an order of magnitude smaller.

We conclude that the stopping criterion is an essential part of the cyclic blow-up algorithm for calculating optimal solutions on real-world instances. In the following we will evaluate the two configurations $C_{\text{strong}}$ and $C_{\text{fast}}$ against existing state-of-the-art solvers.

## 7.3. Comparison With Existing Algorithms

We now compare our two cyclic blow-up configurations and the non-increasing reduction algorithm with other state-of-the art algorithms. For this purpose, we first perform a comparison to the two existing configurations of the branch-and-reduce framework. We therefore analyze the sizes of the reduced graphs, the number of solved instances, and the required solving time. Then we carry out a general comparison with other state-of-the-art algorithms, where we also take into consideration the local searches HILS and DynWVC. Short descriptions of the local searches can be found in Section 3.2. Hereby we investigate the best obtained solution quality and the required calculation time and finally we consider the convergence behavior of the individual algorithms.

For all experiments, we equipped the branch-and-reduce framework with our two cyclic blow-up algorithm configurations $C_{\text{fast}}$ and $C_{\text{strong}}$ from the previous section and our non-increasing algorithm. Accordingly, we call the three solvers Cyclic-Fast, Cyclic-Strong and NonIncreasing in the following. While we always use the different reduction algorithms to compute an initial reduced graph, subsequent Reduce() calls in the branch-and-reduce algorithm during recursion are performed by the non-increasing reduction algorithm. Furthermore, we have replaced the local search of the framework with the hybrid iterated local search (HILS) of Nogueira et al. [43] for our three algorithms. In preliminary experiments we have found that this search finds comparable or better solutions than the previous search within 1 000 iterations and tends to take less time to obtain them.

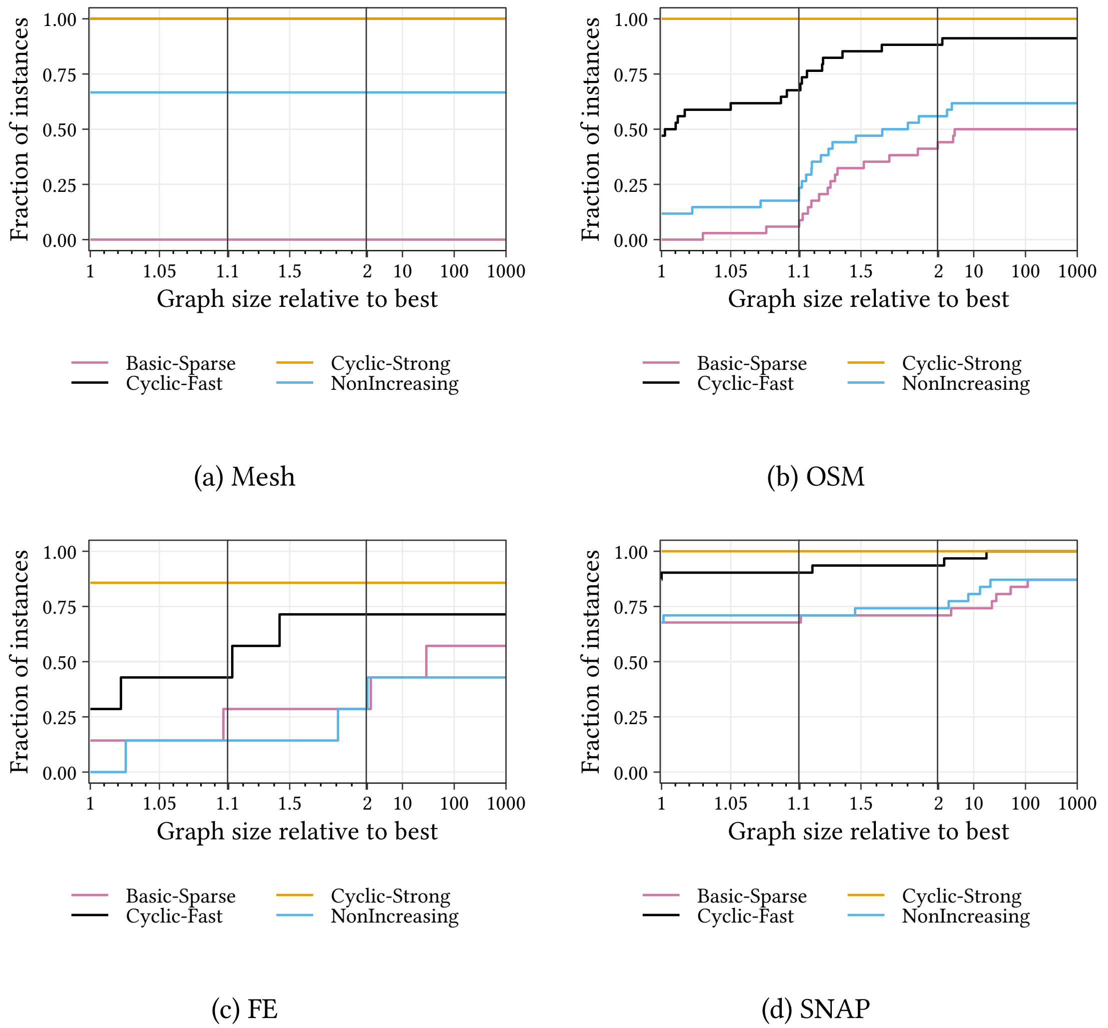



Figure 7.4.: Performance plots of obtained reduced graph sizes for each algorithm on the different instance families.

### 7.3.1. Comparison With Branch-And-Reduce Framework

In the following, a comparison of our new reduction algorithms with the existing branch-and-reduce framework is presented. Our competitors are the two configurations Basic-Sparse and Basic-Dense of the framework as proposed in [33]. While the former always uses all reduction rules as described in Section 4.1, the latter only uses a subset of them. In detail the critical set and clique neighborhood reduction are omitted here. During recursion, the generalized fold reduction is omitted and a simplified clique neighborhood reduction version is used instead, which only takes triangles into account.

Our comparison is divided into a consideration of the initially calculated irreducible graphs and a subsequent evaluation of the number of solved instances and required time.

#### 7.3.1.1. Reduced Graph Sizes

Figure 7.4 shows performance plots for the initial reduced graph sizes calculated for the individual instance families and algorithms. We omit an investigation of Basic-Dense, because it always calculates equal or larger irreducible graphs than Basic-Sparse [33]. Complete tables can be found in the appendix in Tables A.5-A.8. We first note that, with the exception of the `fe_ocean` instance, we can always obtain the smallest reduced graphs by using Cyclic-Strong. For this one instance we get a larger graph than with Basic-Sparse, because here the critical set reduction rule is applied earlier than in the other algorithms. Thus Basic-Sparse is able to obtain an empty reduced graph. The other algorithms change the graph structure by applying additional reduction rules in such a way that the critical set reduction rule is no longer applied as effectively and an empty graph cannot be obtained. However, we find that the Cyclic-Strong algorithm allows us to create much smaller reduced graphs than the original Basic-Sparse algorithm. On the mesh instances, the greatest improvement can be seen, since all graphs obtained with Cyclic-Strong are always empty and Basic-Sparse is not able to obtain an empty graph on a single instance and ends up with reduced graphs of several hundred up to thousands of vertices. While the reduced graphs of Cyclic-Strong and Cyclic-Fast always have the same size on the mesh instances, the quality of Cyclic-Fast decreases slightly on the other instance families. For example, on the OSM instances, Cyclic-Fast calculates a reduced graph of the same size as Cyclic-Strong on only 16 out of 34 instances and can obtain an empty graph on three less instances than Cyclic-Strong. Due to the influence of the weighted critical set reduction described above, the performance profile of the NonIncreasing algorithm on the FE instances always contains the same number of instances or at one less than the one of Basic-Sparse for each quality factor $\tau$. However, on all other families the NonIncreasing algorithm is able to outperform Basic-Sparse. Especially on the mesh instances we achieves much smaller graphs by application of the extended weighted struction, which in about 60% of the instances already results in an empty kernel.

#### 7.3.1.2. Time To Solve

To compare the run times as well as the number of solved instances of the evaluated exact solvers, we present the obtained data in the form of tables with run times of some instances as well as number of solved instances per instance family (see Table 7.5). Our selection focuses on instance on which Basic-Dense or Basic-Sparse are not already able to obtain empty reduced graphs, since we could not measure significant differences between our algorithms on these instances. Full tables can be found in the appendix in Tables A.5-A.8. In addition, we use cactus plots (see Figure 7.5) and performance profile plots (see Figure 7.6) for a graphical visualization of the data in order to obtain information about the run times of the individual algorithms over all instances of a family.

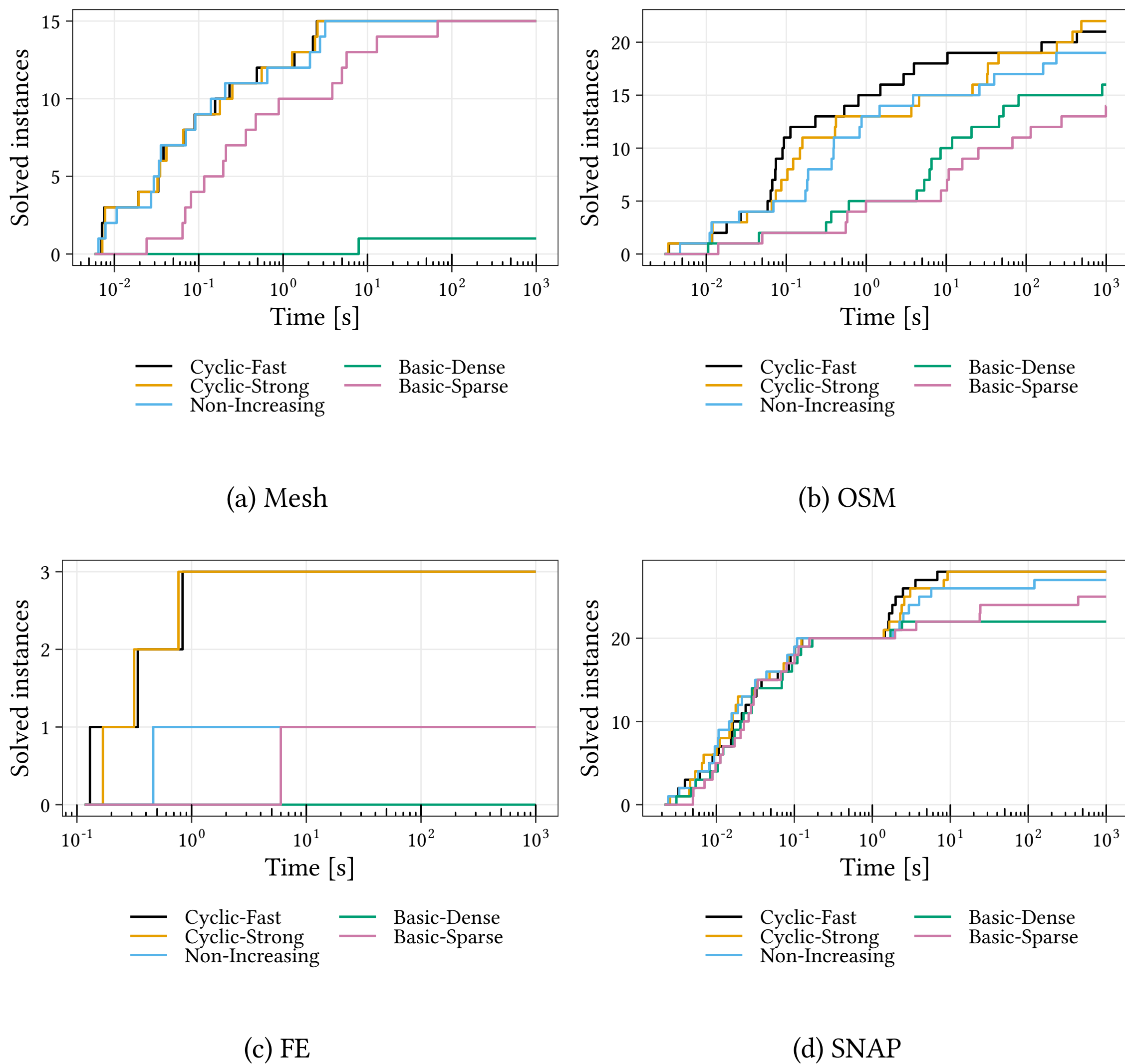
First we want to compare the total number of solved instances. We can see that with our NonIncreasing algorithm we are already able to solve five instances more than by Basic-Sparse and Basic-Dense. We get an increase of five more solvable instances with Cyclic-Fast and finally, Cyclic-Strong is able to solve the OSM instance `north-carolina-AM3`, so we are able to solve 11 more instances than with Basic-Sparse and Basic-Dense. In terms

| Graph | $t_s$ | $t_s$ | $t_s$ | $t_s$ |
|---|---|---|---|---|
| OSM instances | Basic-Dense | NonIncreasing | Cyclic-Fast | Cyclic-Strong |
| district-of-columbia-AM1 | - | 39.81 | **0.80** | 3.66 |
| georgia-AM3 | 892.17 | 25.97 | **10.35** | 32.53 |
| north-carolina-AM3 | - | - | - | **379.09** |
| rhode-island-AM2 | - | 163.07 | **0.53** | 4.58 |
| Solved instances | 47.1% (16/34) | 55.9% (19/34) | 61.8% (21/34) | 64.7% (22/34) |
| SNAP instances | Basic-Sparse | NonIncreasing | Cyclic-Fast | Cyclic-Strong |
| roadNet-TX | 24.30 | 3.98 | **1.64** | 1.65 |
| web-BerkStan | - | 120.05 | **6.83** | 8.25 |
| web-NotreDame | - | - | **1.60** | 2.57 |
| web-Stanford | - | 2.50 | **1.99** | 2.38 |
| Solved instances | 80.6% (25/31) | 87.1% (27/31) | 90.3% (28/31) | 90.3% (28/31) |
| mesh instances | Basic-Sparse | NonIncreasing | Cyclic-Fast | Cyclic-Strong |
| buddha | 67.85 | 2.74 | **2.26** | 2.39 |
| dragon | 3.83 | **0.21** | 0.23 | 0.25 |
| ecat | 12.93 | 3.16 | **2.51** | 2.56 |
| turtle | 4.98 | 0.65 | **0.49** | 0.56 |
| Solved instances | 100.0% (15/15) | 100.0% (15/15) | 100.0% (15/15) | 100.0% (15/15) |
| FE instances | Basic-Sparse | NonIncreasing | Cyclic-Fast | Cyclic-Strong |
| fe_4elt2 | - | - | **0.13** | 0.17 |
| fe_ocean | **5.99** | - | - | - |
| fe_sphere | - | - | 0.83 | **0.77** |
| fe_tooth | - | 0.46 | 0.34 | **0.32** |
| Solved instances | 14.3% (1/7) | 14.3% (1/7) | 42.9% (3/7) | 42.9% (3/7) |

Table 7.5.: Time $t_s$ (in seconds) needed to solve and total solved instances for different exact solvers and instance families. The global best solving time $t_s$ is highlighted in bold.

of absolute number of solved instances, we can report the greatest improvements on the OSM family with six newly solvable instances by Cyclic-Strong. As has already observed by Lamm et al. [33], Basic-Dense performs better on these than Basic-Sparse and is able to solve 16 out of 34 instances. With our Cyclic-Strong algorithm we are now able to solve 22 of 34 instances, corresponding to 17.6% more solvable instances. On the SNAP instances we can see that we can solve all web graphs now and thus three more instances become feasible by Cyclic-Fast and Cyclic-Strong. Only two social networks and the instance as-skitter of the SNAP family are still not solvable within the time limit of 1 000 seconds. While we are already able to solve all instances of the mesh family by Basic-Sparse, two more FE instances can be solved by Cyclic-Fast and Cyclic-Strong than by Basic-Sparse. It is interesting to note that we actually can solve three new instances here. However, the `fe_ocean` instance becomes infeasible for our algorithms since we obtain a reduced graph with more than 100 000 vertices and Basic-Dense generates an empty reduced graph as described in the previous section.

Comparing the time that our algorithms require to solve the instances with the two methods Basic-Sparse and Basic-Dense, we can see improvements on almost all instances. Our Cyclic-Fast algorithm finds solutions on five mesh instances, 13 OSM instances and three SNAP instances by a factor of ten faster than Basic-Sparse and Basic-Dense, on the two OSM instances `pennsylvania-AM3` and `utah-AM3` as well as `roadNet-CA` of the SNAP family we even find solutions faster by two orders of magnitude. We can explain these large differences by two reasons. First, we have already noticed in the previous section that Cyclic-Fast calculates much smaller reduced graphs than Basic-Sparse



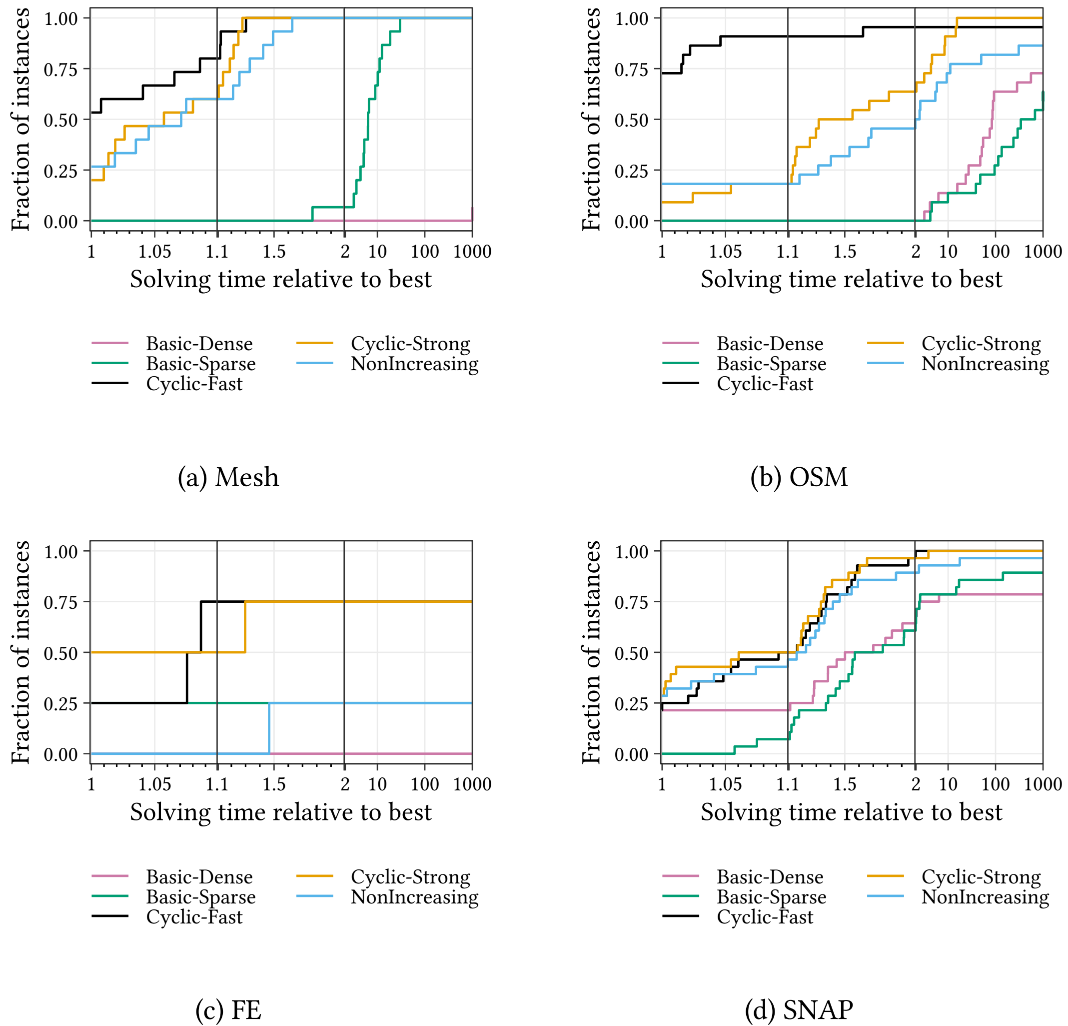

(a) Mesh (b) OSM

(c) FE (d) SNAP

Figure 7.5.: Cactus plots for the different instance families and evaluated solvers.

and Basic-Dense, so the actual solution time tends to be shorter. Furthermore, we have also already seen in section 7.2.1.2 that the generalized fold reduction rule has a very negative influence on the runtime of the non-increasing reduction algorithm. While all our algorithms Cyclic-Fast, Cyclic-Strong and NonIncreasing do not use this rule, it is applied in both configurations Basic-Sparse and Basic-Dense. This results in reduction times for the Cyclic-Fast algorithm that are up to several orders of magnitude lower than for Basic-Sparse and Basic-Dense on many instances even though we have a certain overhead by the cyclic blow-up algorithm (see Tables A.5-A.8).

Looking at the cactus plots from Figure 7.5, we find that our three algorithms can always solve more instances over time than Basic-Sparse and Basic-Dense. In particular, we can see great improvements on the OSM instances: Our Cyclic-Fast algorithm for instance can solve 19 of its 21 solved instances within ten seconds and thus performs better




(a) Mesh

(b) OSM

(c) FE

(d) SNAP

Figure 7.6.: Performance plots of solving time for each algorithm on the different instance families.

than Cyclic-Strong since it has only solved 15 instances at this time. If we compare Cyclic-Fast with the two existing algorithms Basic-Sparse and Basic-Dense, we see that after one second with Cyclic-Fast we already have solved 15 instances, which is three times as many as Basic-Sparse and Basic-Dense. Finally, the algorithm Basic-Sparse is always able to solve more instances than Basic-Dense at any time. If we look at the NonIncreasing algorithm, it has fewer instances solved than Cyclic-Fast at all times and is outperformed outside the one to ten second interval by Cyclic-Strong.

Regarding the SNAP instances, we can hardly see any differences between the different solvers within the first second. This is due to the fact that already on 20 of the 31 instance can be reduced by the reduction rule set of Basic-Dense to graphs with less than ten vertices, namely all peer-to-peer networks, collaboration networks, communication networks and three out of five social networks. Fundamental differences are due to the

newly solvable web-crawl instances and shorter run times for the road net instances by Cyclic-Fast and Cyclic-Strong.

For the mesh family there are almost no differences between our three algorithms. Compared to Basic-Sparse, we can state that we are always able to solve at least two or more instances at the same time within the first ten seconds using one of our three algorithms. We also note that Basic-Dense can only solve one instance of this family. This is especially interesting because Basic-Dense can solve more OSM instances than Basic-Sparse, but in particular our two algorithms Cyclic-Strong and Cyclic-Fast always get better results on all families than both algorithms.

If we look at the performance profile plots in Figure 7.6 we can make further statements especially for the quality between our three algorithms. This shows that with Cyclic-Fast we can solve more than half of all mesh instances and about 75% of all OSM instances at the same speed or faster than any other algorithm. In contrast, Basic-Sparse can solve only one instance of the mesh and OSM families in less than twice the time of the best solver. Although Cyclic-Strong can solve one OSM instance more than Cyclic-Fast, it only solves about 10% of all OSM instances the fastest and takes at least 1.5 times the time on half of all instances to solve.

### 7.3.2. Comparison With State-Of-The-Art Algorithms

In the following we provide a comparison of our algorithms with state-of-the-art solvers for the maximum weight independent set problem. Besides the two configurations Basic-Sparse and Basic-Dense of the branch-and-reduce framework we compare our algorithms with the two local searches DynWVC and HILS. For the former we use both configurations DynWVC1 and DynWVC2 described by Cai et al. [12]. We compare both the best achievable solutions of the different methods as well as the convergence behavior regarding the solution quality.

#### 7.3.2.1. Best Solution Quality And Time

In Table 7.6 an overview of the results of our comparison with heuristic methods is given.

We list the maximum obtained weights $w_{max}$ of each algorithm over the five runs with different random seeds and the time $t_{min}$ to obtain them. If several solutions with maximum weight $w_{max}$ were found in different runs, $t_{max}$ corresponds to the minimum time to obtain them. The global best solution among all algorithms for each instance is highlighted in bold. Furthermore, for our exact algorithms the number of optimal solved instances is shown, while for heuristic methods the percentage of exactly solved instances is given, on which they can also find an optimal solution. Finally, lines are highlighted in gray if one of our two configurations Cyclic-Fast or Cyclic-Strong is able to solve the corresponding instance. For the individual instance families, we always list only either DynWVC1 or DynWVC2 depending on which of the two configurations provides better performance. We omit a listing of Basic-Sparse, Basic-Dense and NonIncreasing, since as seen before, Cyclic-Fast and Cyclic-Strong outperform these in terms of reduced graph size, number of solved instances and solving time. For full tables we refer the reader to the Tables A.9-A.12 in the appendix.

| Graph | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ |
|---|---|---|---|---|---|---|---|---|
| OSM instances | DynWVC2 | | HILS | | Cyclic-Fast | | Cyclic-Strong | |
| alabama-AM2 | 0.24 | 174269 | 0.03 | **174309** | 0.01 | **174309** | 0.01 | **174309** |
| district-of-columbia-AM2 | 915.18 | 208977 | 400.69 | **209132** | 4.21 | **209132** | 84.21 | 209131 |
| florida-AM3 | 862.04 | 237120 | 3.98 | **237333** | 1.57 | **237333** | 40.97 | **237333** |
| georgia-AM3 | 1.31 | **222652** | 0.04 | **222652** | 0.98 | **222652** | 12.97 | **222652** |
| greenland-AM3 | 640.46 | 14010 | 1.18 | **14011** | 10.95 | **14011** | 58.24 | 14008 |
| new-hampshire-AM3 | 1.63 | **116060** | 0.03 | **116060** | 0.05 | **116060** | 0.08 | **116060** |
| rhode-island-AM2 | 13.90 | 184576 | 0.24 | **184596** | 0.41 | **184596** | 4.37 | **184596** |
| utah-AM3 | 136.90 | **98847** | 0.07 | **98847** | 0.09 | **98847** | 0.27 | **98847** |
| Solved instances | | | | | | 61.8% (21/34) | | 64.7% (22/34) |
| Optimal weight | | 68.2% (15/22) | | 100.0% (22/22) | | | | |
| SNAP instances | DynWVC2 | | HILS | | Cyclic-Fast | | Cyclic-Strong | |
| as-skitter | 383.97 | 123273938 | 999.32 | 122658804 | 346.69 | 124137148 | 354.71 | **124137365** |
| ca-AstroPh | 125.05 | 797480 | 13.47 | **797510** | 0.02 | **797510** | 0.02 | **797510** |
| email-EuAll | 132.62 | **25286322** | 338.14 | **25286322** | 0.07 | **25286322** | 0.07 | **25286322** |
| p2p-Gnutella06 | 186.97 | 548611 | 1.29 | **548612** | 0.01 | **548612** | 0.01 | **548612** |
| roadNet-PA | 469.18 | 60990177 | 999.94 | 60037011 | 0.96 | **61731589** | 1.04 | **61731589** |
| soc-LiveJournal1 | 999.99 | 279231875 | 1000.00 | 255079926 | 51.33 | 284036222 | 44.19 | **284036239** |
| web-Google | 324.65 | 56206250 | 995.92 | 56008278 | 1.72 | **56326504** | 6.44 | **56326504** |
| wiki-Vote | 0.32 | **500079** | 10.34 | **500079** | 0.02 | **500079** | 0.02 | **500079** |
| Solved instances | | | | | | 90.3% (28/31) | | 90.3% (28/31) |
| Optimal weight | | 28.6% (8/28) | | 57.1% (16/28) | | | | |
| mesh instances | DynWVC2 | | HILS | | Cyclic-Fast | | Cyclic-Strong | |
| buddha | 797.35 | 56757052 | 999.94 | 55490134 | 1.75 | **57555880** | 1.77 | **57555880** |
| dragon | 981.51 | 7944042 | 996.01 | 7940422 | 0.21 | **7956530** | 0.22 | **7956530** |
| ecat | 542.87 | 36129804 | 999.91 | 35512644 | 2.19 | **36650298** | 2.29 | **36650298** |
| Solved instances | | | | | | 100.0% (15/15) | | 100.0% (15/15) |
| Optimal weight | | 0.0% (0/15) | | 0.0% (0/15) | | | | |
| FE instances | DynWVC1 | | HILS | | Cyclic-Fast | | Cyclic-Strong | |
| fe_ocean | 983.53 | **7222521** | 999.57 | 7069279 | 18.85 | 6591832 | 19.04 | 6591537 |
| fe_sphere | 875.87 | 616978 | 843.67 | 616528 | 0.63 | **617816** | 0.67 | **617816** |
| Solved instances | | | | | | 42.9% (3/7) | | 42.9% (3/7) |
| Optimal weight | | 0.0% (0/3) | | 0.0% (0/3) | | | | |

Table 7.6.: Best solution found by each algorithm and time (in seconds) required to compute it. The global best solution is highlighted in bold. Rows are highlighted in gray if one of our exact solvers is able to solve the corresponding instances.

Considering the OSM family, we can see that HILS calculates optimal solutions on all 22 of the 34 instances that can be solved by our algorithm Cyclic-Strong. In contrast, DynWVC2 can find an optimal solution on 15 of 22 instances which is one more than DynWVC1. On the 12 remaining instances on which our algorithms Cyclic-Fast and Cyclic-Strong obtain solutions whose optimality they cannot prove during the time limit, HILS is able to calculate the best solution among all algorithms on ten instances. Cyclic-Strong on the other hand can obtain a globally best solution on two of these unsolved instances, while Cyclic-Fast computes best solutions on four further instances. On two of these instances, the solution is even better than the solution found by HILS. Finally, both DynWVC configurations can only compute a global best solution on the unsolved instance idaho-AM3, which is also found by HILS. In direct comparison, HILS can get better solutions than Cyclic-Fast on five unsolved instances, while conversely Cyclic-Fast can compute better solutions than HILS on two instances. If we compare the required run times of the individual algorithms to obtain their best solutions, we find that HILS performs better than the other algorithms. On all 17 instances on which one of the DynWVC configurations and HILS compute solutions of the same weight,

HILS needs less time to obtain them, which is at least one order of magnitude less on most instances. In contrast, on the 27 instances where Cyclic-Fast and HILS calculate solutions of the same weight, HILS is faster on 19 instances, on seven instances even by one order of magnitude. Conversely, Cyclic-Fast obtains the solutions on eight instances faster than HILS and on the `district-of-columbia-AM2` instance even by almost more than two orders of magnitude. Altogether we can state that the solutions found by the local searches are always very close to each other and mostly differ by less than 0.1%. However, the solution quality of our two algorithms Cyclic-Fast and Cyclic-Strong on the unsolved instances exhibits a much greater difference in some cases. Here, we can observe a certain connection to the reduced graph size found by Cyclic-Fast and Cyclic-Strong (see Table A.7). So we obtain solutions on the `district-of-columbia-AM3` instance which have only about 60% of the weight of the best solution found by HILS. At the same time, the reduced graphs of both algorithms have more than 25 000 vertices. The reason for this behavior is the initial solution of the local search, which turns out to be very poor within 1 000 iterations on graphs of that size. Since this solution also serves as a bad lower bound, it cannot be further improved in the subsequent branch-and-reduce process.

If we look at the SNAP instances, we get a similar picture as Lamm et al. [33] with its configurations Basic-Sparse and Basic-Dense. In the previous section we have already seen that Cyclic-Fast and Cyclic-Strong can solve 28 of the 31 instances optimally. In contrast, HILS can only calculate optimal solutions on 16 of these 28 instances. Furthermore, DynWVC2 performs better than DynWVC1 and is able to obtain optimal solutions on eight of the solved instances. Looking at the three unsolved instances, Cyclic-Strong computes the best solution on `as-skitter` and `soc-LiveJournal1`, while DynWVC1 obtains it on `soc-pokec-relationships`. In a direct comparison of the local searches we can state that HILS finds the better solutions on peer-to-peer networks, collaboration networks and communication networks as DynWVC2 and needs less time to obtain them. Conversely, DynWVC2 is superior to HILS on road networks and web-crawl instances both in terms of quality and runtime. Overall, we recognize that both DynWVC1 and HILS are inferior in terms of runtime to our algorithms Cyclic-Fast and Cyclic-Strong and exhibit several orders of magnitude higher run times for calculating their best solutions. Since the reduced graphs of both algorithms Cyclic-Fast and Cyclic-Strong on the `soc-pokec-relationships` have several million vertices, this is the only instance on which the best solutions of our algorithms are inferior to the ones of the local searches.

On the mesh instances, we can observe a similar pattern to the SNAP instances. Our algorithms Cyclic-Fast and Cyclic-Strong are able to solve all instances optimally and always need less than three seconds to obtain them. On the other hand, none of the evaluated local searches is able to compute an optimal solution on a single instance and require run times which are several orders of magnitude higher than those of our algorithms.

Finally, on the FE family we can conclude that again neither DynWVC nor HILS are able to obtain a solution of equal weight on any of the three solved instances by Cyclic-Fast and Cyclic-Strong. However, considering the unsolved instances, the `fe_body` instance is the only one on which Cyclic-Fast calculates the global best solution since the reduced graph only consists of a few hundred vertices. On all remaining instances, one of the two

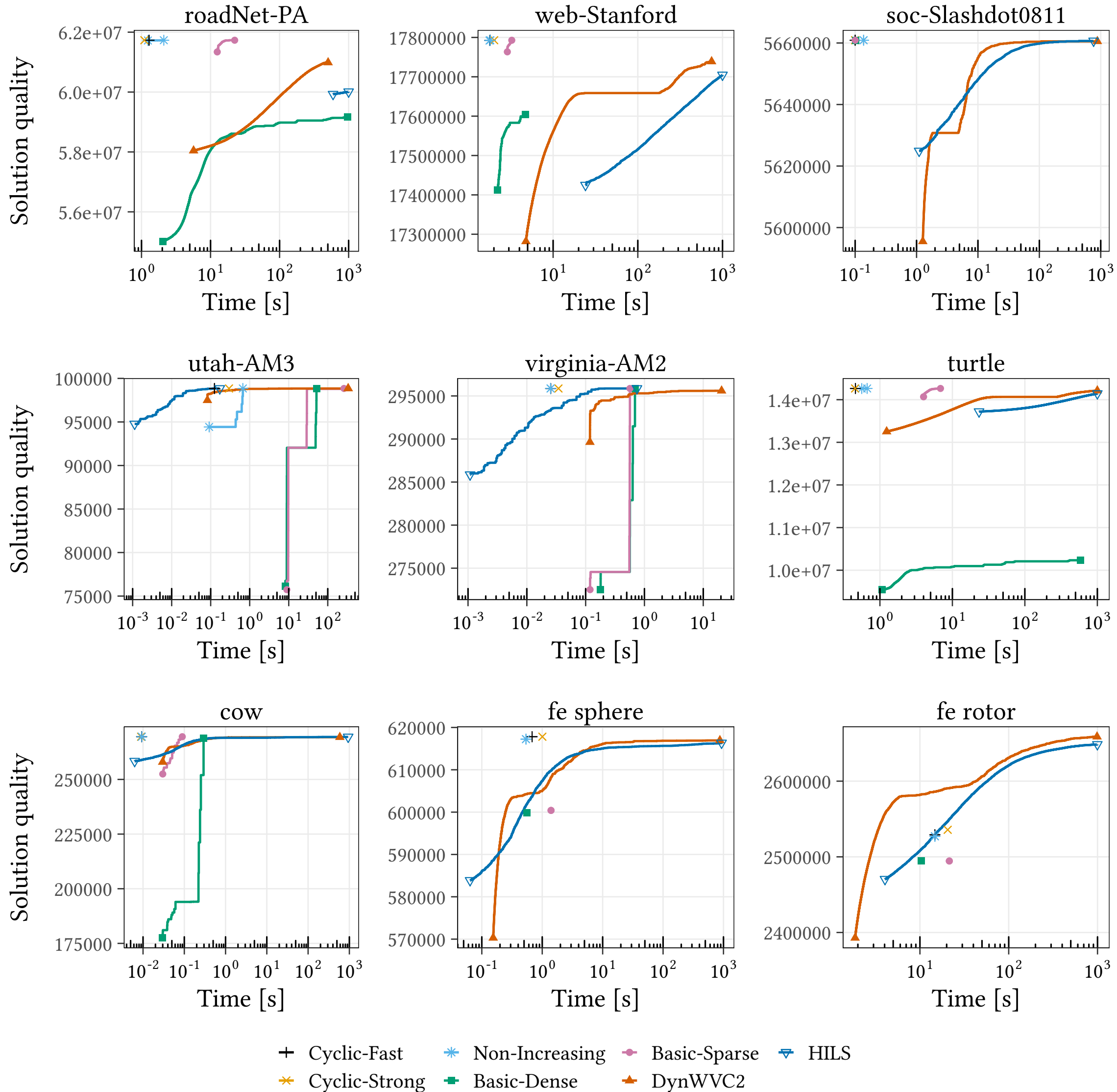



Figure 7.7.: Solution quality over time for three SNAP instances (upper row), two OSM instances (`utah-AM3`, `virginia-AM2`), two mesh instances (`turtle`, `cow`) and two fe instances (`fe_sphere`, `fe_rotor`).

DynWVC configurations calculates the best solution, because our algorithms fail to reduce them enough and yield reduced graphs with several 10 000 vertices. In a direct comparison of the two DynWVC configurations and HILS we see that DynWVC can obtain better solutions on five of the seven instances, and that it takes less time except for the `fe_sphere` instance. However, HILS is able to get better solutions on the `fe_4elt2` and `fe_tooth` instances than both DynWVC configurations and takes less time on the `fe_4elt2` instance.

#### 7.3.2.2. Solution Quality Convergence

In order to make a more detailed statement for the evaluated solvers about the solution quality over time, we present convergence plots for some instances in Figure 7.7. Further convergence plots for other instances can be found in the appendix in Figures A.7-A.9. Any convergence plot was therefore accumulated over five runs with different random seeds using event-based geometric means. Since DynWVC2 and DynWVC1 exhibit almost the same convergence behavior, only DynWVC2 is shown in the figures for the sake of simplicity. Furthermore, we always reference both algorithms as DynWVC. Depending on the instance family, we can observe very different convergence patterns of the individual solvers, which we will briefly examine in the following.

If we take a look at the OSM instances, we see that the local search HILS always gets an initial solution fastest of all algorithms and converges to optimal solutions much faster than DynWVC. For almost every instance, the current solution of HILS is always as good as that of DynWVC or better. In comparison to all exact methods, the local searches are usually able to output high-quality initial solutions much faster. If we take a closer look at our algorithm Cyclic-Fast, it finds an initial solution on about two thirds of the instances later than DynWVC, however, this solution is usually better than the current best solution of DynWVC. On the remaining instances, Cyclic-Fast finds an initial solution faster than DynWVC, which is already optimal in most cases. Compared to the other algorithms, Cyclic-Strong always needs more time to output a first solution than Cyclic-Fast, due to the increased overhead of the initial reduction, while NonIncreasing computes an initial solution earlier on some instances. In comparison to the two exact algorithms Basic-Sparse and Basic-Dense we can see that we often need considerably less time to calculate an initial solution with Cyclic-Fast and that this solution always turns out to be as good or better than the ones obtained by Basic-Dense and Basic-Sparse. We can explain this behavior by the smaller reduction times of Cyclic-Fast in comparison to Basic-Sparse and Basic-Dense, caused by the application of the generalized fold reduction rule, that we already observed in Section 7.3.1.2. For example, on the instance `utah-AM3`, it takes Basic-Sparse and Basic-Dense with 8.2 seconds a factor of 70 longer than Cyclic-Fast to compute an initial solution that has also only about 77% of the weight of the (optimal) initial solution found byCyclic-Fast. Even after nine seconds, the weight of the best found solution by Basic-Sparse and Basic-Dense is still only about 93% of the weight of an optimal solution. Another aspect that we can observe is that in about two thirds of all instances for Cyclic-Fast and Cyclic-Strong, the initial solution already corresponds to their overall best solution. For the NonIncreasing algorithm this is only the case in about one third of the instances, while Basic-Sparse and Basic-Dense usually have several intermediate solutions until they obtain their optimal solution. We can explain this by the sizes of the irreducible graphs, which are larger for the latter algorithms. Especially for Basic-Sparse and Basic-Dense we could observe in preliminary experiments that their obtained irreducible graphs consist of several large components, which explains the large steps in convergence behavior.

Regarding the SNAP family, we can state that our algorithms Cyclic-Fast and NonIncreasing are able to output an initial solution faster than the local searches on almost every instance and at the same time they are comparable even better than the

overall best found solution of the local searches. The only exceptions are the instances `soc-pokec-relationships` and `wiki-Vote`. On the `wiki-Vote` instance, HILS can output an initial solution earlier, but the initial solutions of our algorithms then dominate both local searches. Looking at the `soc-pokec-relationships` instance, DynWVC can find both the first and overall best solution and has at any time a better current solution than our approaches. In a direct comparison of the two local searches we see a certain pattern on peer-to-peer networks, collaboration networks, communication networks and social networks. With the exception of the instances `soc-pokec-relationships` and `soc-LiveJournal1`, HILS is always able to output an initial solution faster than DynWVC. Subsequently, DynWVC is either able to obtain a better current solution than HILS by its initial solution or it catches up with HILS on many instances over time. While HILS is ultimately able to outperform DynWVC on all communication networks with better solutions, there is no clear result on the remaining instances, since both algorithms find very similar solutions over time. On the road networks and web-crawl instances, however, we can clearly see that DynWVC outputs initial solutions faster and always has a better current solution than HILS, thus DynWVC outperforms HILS on these instances. Comparing our algorithms to the exact solution methods Basic-Sparse and Basic-Dense we can observe either the same convergence behavior or a better one. Especially on the web-crawl and road network instances all of our algorithms can find initial solutions with better weight faster than Basic-Sparse and Basic-Dense. Furthermore, the current best solutions of Cyclic-Fast, Cyclic-Strong and NonIncreasing are always better than those of the other two exact methods. If we take a closer look at the road network instance `roadNet-PA`, for example, we see that our two algorithms Cyclic-Fast and Cyclic-Strong have already obtained an optimal solution after about one second. At the same time NonIncreasing finds an initial solution, which it can improve to an optimal solution after a total of two seconds. On the other hand, we see that the algorithm Basic-Sparse also outputs an initial solution after two seconds, but its weight is only 90% of the optimum. After five seconds DynWVC is finally able to output an initial solution that is better than the current solution of Basic-Dense. During the time interval between ten and thirty seconds, Basic-Sparse has a better current solution than DynWVC, but then DynWVC can beat Basic-Dense in terms of solution quality until the time limit is reached. Because of the increased reduction overhead of Basic-Sparse compared to Basic-Dense, the algorithm Basic-Sparse needs about one order of magnitude more time to output its first solution than Cyclic-Fast, which is about 1% away from the optimum. In total Basic-Sparse needs about a factor 20 longer to get an optimal solution than Cyclic-Fast. Finally, we see that HILS needs just under 600 seconds to output an initial solution whose weight is only 97% of the optimal solution and cannot be increased significantly until the time limit is reached.

For the mesh family we can also see a clear advantage of our algorithms compared to the state-of-the-art solvers. On almost every instance, Cyclic-Fast and Cyclic-Strong always compute an initial solution the fastest, which is also always optimal. Only on the instances `cow` and `venus`, HILS outputs an initial solution prior to the other algorithms. However, both local searches do not achieve an optimal solution within the time limit on any instance. In direct comparison, HILS is outperformed on about half of the instances by DynWVC, because DynWVC obtains its initial solutions faster and always has a better current solution until the end. On the remaining instances, HILS can initially output

solutions faster, but is soon caught up by DynWVC until both algorithms have nearly equal current best solutions. In comparison, Basic-Sparse always takes longer to compute an initial solution except for the instance gameguy, but then converges to the optimal solution in a short time. On the other hand, Basic-Sparse is able to calculate an initial solution earlier due to the shorter reduction time, but due to the large size of the graph it is much worse and converges to an optimal solution only on the instance beethoven.

When looking at the FE family, we find a quite heterogeneous situation. While our configuration Cyclic-Fast can obtain initial solutions on four of the seven instances within one second, which are always better than the best solutions of both local searches and the exact methods Basic-Sparse and Basic-Dense, on the other instances, Cyclic-Fast requires more time to compute much worse initial solutions. For example, on the instance fe_ocean, the calculation of an initial solution takes about 20 seconds and cannot be improved further, so the best solution of Cyclic-Fast is about 10% away from the optimum. As already mentioned in Section 7.3.1.1, Basic-Sparse can obtain an empty reduced graph on this instance due to a different reduction rule order and thus finds an optimal solution. Even though Basic-Dense can often compute initial solutions faster than our algorithms, they are much worse in quality, so our algorithms can outperform both Basic-Sparse and Basic-Dense on all other instances by providing better solutions. In a direct comparison of the two local searches, HILS is outperformed by DynWVC. On the two instances fe_ocean and fe_rotor, DynWVC both obtains initial solutions faster and has better current solutions than HILS of time. From the moment when both searches have generated an initial solution on the instance fe_ocean, the strongest deviation of the solutions from HILS compared to DynWVC takes 5%. On the two instances fe_body and fe_tooth, DynWVC can initially get better solutions until both searches finally get solutions of almost the same weight. On the last three instances HILS and DynWVC alternate several times in calculating the better solution among each other until DynWVC can finally slightly dominate HILS.

# 8. Discussion

## 8.1. Conclusion

In this thesis we have developed new effective data reduction techniques for the maximum weight independent set problem using the weighted struction. Through a theoretical analysis we have shown that two of the struction variants are very powerful reduction rules and already subsume six of the eight reduction rules of the state-of-the-art algorithm by Lamm et al. [33]. By distinguishing between three different classes of structions, we then developed two new reduction algorithms based on the existing reduction rule set of Lamm et al. [33]. While our non-increasing reduction algorithm uses decreasing and plateau structions, i.e. structions that decrease the number of vertices or it the same, to reduce the input graph iteratively, our cyclic blow-up algorithm uses increasing structions to exploit the full potential of the struction, by also allowing struction applications that blow-up the graph. Although these structions initially increase the number of vertices, they can also allow further application of reduction rules, potentially resulting in an overall decrease of the graph size.

Through our experimental evaluation on real-world instances, we could finally show that our reduction algorithms are not only advantageous in theory but are also very effective in practice. By only using the extended weighted struction or the extended reduced weighted struction we were able to obtain smaller irreducible graphs on a large number of instances compared to previous methods. By combining these structions with already existing reduction rules we were able to reduce the obtained graph sizes even further by our non-increasing reduction algorithm. Finally, we were able to show that our approach of the cyclic blow-up algorithm also works well in practice, since it ultimately enabled us to obtain considerably smaller reduced graphs than all algorithms mentioned previously. All in all, on some instances we able to completely reduce previously irreducible graphs with several thousand vertices to an empty graph or to obtain reduced graphs which are up to two orders of magnitude smaller.

During the parameter tuning, where we integrated the cyclic blow-up algorithm into the branch-and-reduce framework of Lamm et al. [33], we could determine two different configurations. While our strong configuration now allows us to solve about 12.5% more of the evaluated instances than existing approaches, we are also able by our fast configuration to reduce the geometric mean of the run times of all previously solvable instances by a factor of five.

Nevertheless, we have also seen that some instances still cannot be solved by branch-and-reduce algorithms, since their reduced graphs still have several thousand up to millions of vertices. By further research on even more advanced data reduction techniques some or even all of these instances might become feasible in the future.

## 8.2. Future Work

In the description of the cyclic blow-up algorithm we have seen that it has a large number of adjustable components. Our current configurations Cyclic-Fast and Cyclic-Strong do not use the full potential of the cyclic blow-up algorithm yet and for instance always reject new graphs if they are larger than the current graph. At this point, we want to investigate in the future to what extent progress can be achieved by taking a middle ground between always accept and only accept if the graph is smaller than before. This could lead both to runtime advantages, since backtracking would have to be done less frequently, and to potentially smaller reduced graphs. In this context, a new quality criterion, that measures the hardness of a graph in terms of the solvability by (exact) methods, could be developed. Ideas that emerged during this work would involve the convergence behavior of local searches on the graph. Faster convergence would then intuitively indicate a simpler to solve graph than vice versa.

Also, the cyclic blow-up algorithm could be extended to handle a multitude of reduced graphs as individuals, similar to evolutionary algorithms. Mutations would thus be performed by applying increasing structions and other reduction rules, while for the fitness function either the graph size or the convergence behavior as described above would be taken into account.

Now that we have seen that generalized reduction rules can yield great progress in terms of reduced graph size, further research in this area could also lead to considerable success. For instance, existing rules of the unweighted case like conic reduction [39] or clique reduction [38] could be investigated for the weighted case and integrated into the cyclic blow-up algorithm. Alternatively, new techniques are also conceivable, which could, for example, extend the twin or domination reduction, which are not subsumed by the struction.

Finally, in a larger context, the basic idea of the cyclic blow-up algorithm could be applied to other $\mathcal{NP}$-complete problems where data reduction techniques are already used, such as the dominating set or minimum cut problem.

# A. Appendix

## A.1. Basic Graph Property Tables

| Graph | $\|V\|$ | \|E\| |
|---|---:|---:|
| fe_4elt2 | 11143 | 65636 |
| fe_body | 45087 | 327468 |
| fe_ocean | 143437 | 819186 |
| fe_pwt | 36519 | 289588 |
| fe_rotor | 99617 | 1324862 |
| fe_sphere | 16386 | 98304 |
| fe_tooth | 78136 | 905182 |

Table A.1.: Basic properties of FE instances

| Graph | $\|V\|$ | \|E\| |
|---|---:|---:|
| beethoven | 4419 | 12982 |
| blob | 16068 | 48204 |
| buddha | 1087716 | 3263148 |
| bunny | 68790 | 206034 |
| cow | 5036 | 14732 |
| dragon | 150000 | 450000 |
| dragonsub | 600000 | 1800000 |
| ecat | 684496 | 2053488 |
| face | 22871 | 68108 |
| fandisk | 8634 | 25636 |
| feline | 41262 | 123786 |
| gameguy | 42623 | 127700 |
| gargoyle | 20000 | 60000 |
| turtle | 267534 | 802356 |
| venus | 5672 | 17016 |

Table A.2.: Basic properties of mesh instances

| Graph | $\lvert V \rvert$ | \|E\| |
|---|---:|---:|
| as-skitter | 1696415 | 22190596 |
| ca-AstroPh | 18772 | 396100 |
| ca-CondMat | 23133 | 186878 |
| ca-GrQc | 5242 | 28968 |
| ca-HepPh | 12008 | 236978 |
| ca-HepTh | 9877 | 51946 |
| email-Enron | 36692 | 367662 |
| email-EuAll | 265214 | 728962 |
| p2p-Gnutella04 | 10876 | 79988 |
| p2p-Gnutella05 | 8846 | 63678 |
| p2p-Gnutella06 | 8717 | 63050 |
| p2p-Gnutella08 | 6301 | 41554 |
| p2p-Gnutella09 | 8114 | 52026 |
| p2p-Gnutella24 | 26518 | 130738 |
| p2p-Gnutella25 | 22687 | 109410 |
| p2p-Gnutella30 | 36682 | 176656 |
| p2p-Gnutella31 | 62586 | 295784 |
| roadNet-CA | 1965206 | 5533214 |
| roadNet-PA | 1088092 | 3083796 |
| roadNet-TX | 1379917 | 3843320 |
| soc-Epinions1 | 75879 | 811480 |
| soc-LiveJournal1 | 4847571 | 85702474 |
| soc-Slashdot0811 | 77360 | 938360 |
| soc-Slashdot0902 | 82168 | 1008460 |
| soc-pokec-relationships | 1632803 | 44603928 |
| web-BerkStan | 685230 | 13298940 |
| web-Google | 875713 | 8644102 |
| web-NotreDame | 325729 | 2180216 |
| web-Stanford | 281903 | 3985272 |
| wiki-Talk | 2394385 | 9319130 |
| wiki-Vote | 7115 | 201524 |

Table A.3.: Basic properties of SNAP instances

| Graph | $\lvert V \rvert$ | \|E\| |
|---|---:|---:|
| alabama-AM2 | 1164 | 38772 |
| alabama-AM3 | 3504 | 619328 |
| district-of-columbia-AM1 | 2500 | 49302 |
| district-of-columbia-AM2 | 13597 | 3219590 |
| district-of-columbia-AM3 | 46221 | 55458274 |
| florida-AM2 | 1254 | 33872 |
| florida-AM3 | 2985 | 308086 |
| georgia-AM3 | 1680 | 148252 |
| greenland-AM3 | 4986 | 7304722 |
| hawaii-AM2 | 2875 | 530316 |
| hawaii-AM3 | 28006 | 98889842 |
| idaho-AM3 | 4064 | 7848160 |
| kansas-AM3 | 2732 | 1613824 |
| kentucky-AM2 | 2453 | 1286856 |
| kentucky-AM3 | 19095 | 119067260 |
| louisiana-AM3 | 1162 | 74154 |
| maryland-AM3 | 1018 | 190830 |
| massachusetts-AM2 | 1339 | 70898 |
| massachusetts-AM3 | 3703 | 1102982 |
| mexico-AM3 | 1096 | 94262 |
| new-hampshire-AM3 | 1107 | 36042 |
| north-carolina-AM3 | 1557 | 473478 |
| oregon-AM2 | 1325 | 115034 |
| oregon-AM3 | 5588 | 5825402 |
| pennsylvania-AM3 | 1148 | 52928 |
| rhode-island-AM2 | 2866 | 590976 |
| rhode-island-AM3 | 15124 | 25244438 |
| utah-AM3 | 1339 | 85744 |
| vermont-AM3 | 3436 | 2272328 |
| virginia-AM2 | 2279 | 120080 |
| virginia-AM3 | 6185 | 1331806 |
| washington-AM2 | 3025 | 304898 |
| washington-AM3 | 10022 | 4692426 |
| west-virginia-AM3 | 1185 | 251240 |

Table A.4.: Basic properties of OSM instances

## A.2. Reduced Graph Size Convergence Plots

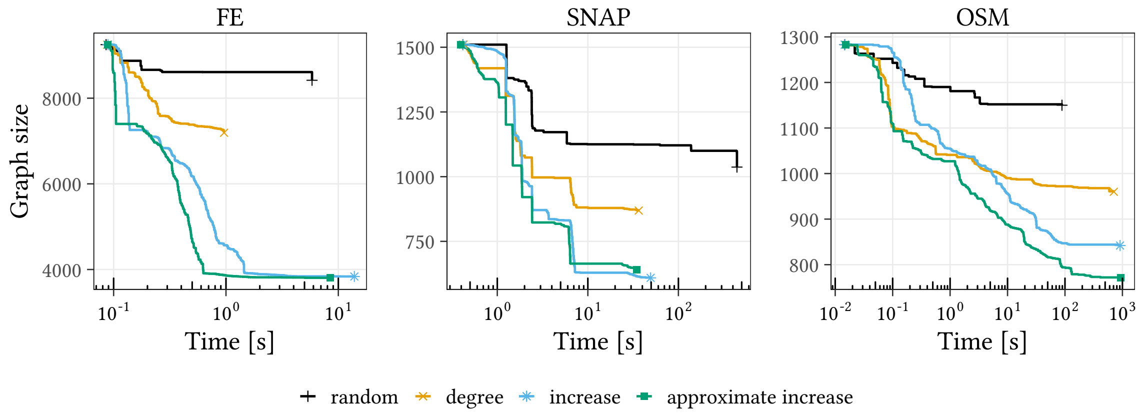



Figure A.1.: Accumulated convergence plots for configuration $C_{\text{accept}}$ with different vertex selection strategies.

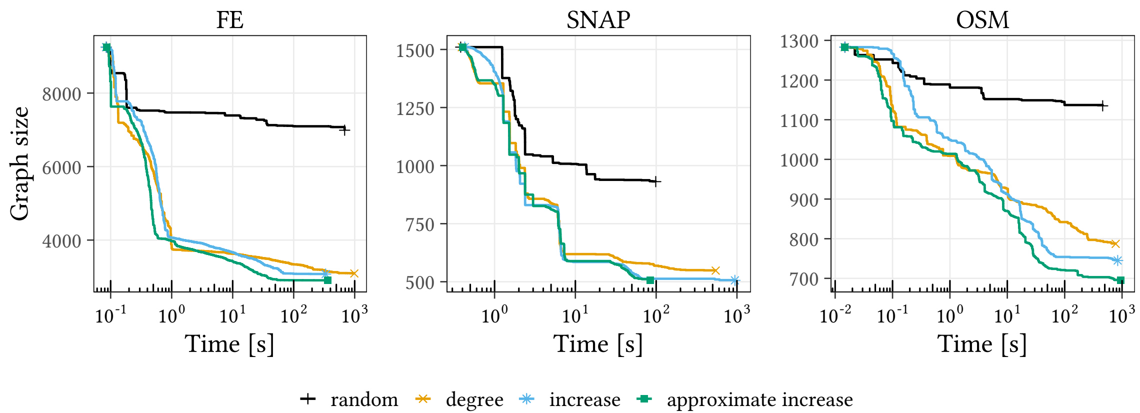



Figure A.2.: Accumulated convergence plots for configuration $C_{\text{tabu}}$ with different vertex selection strategies.

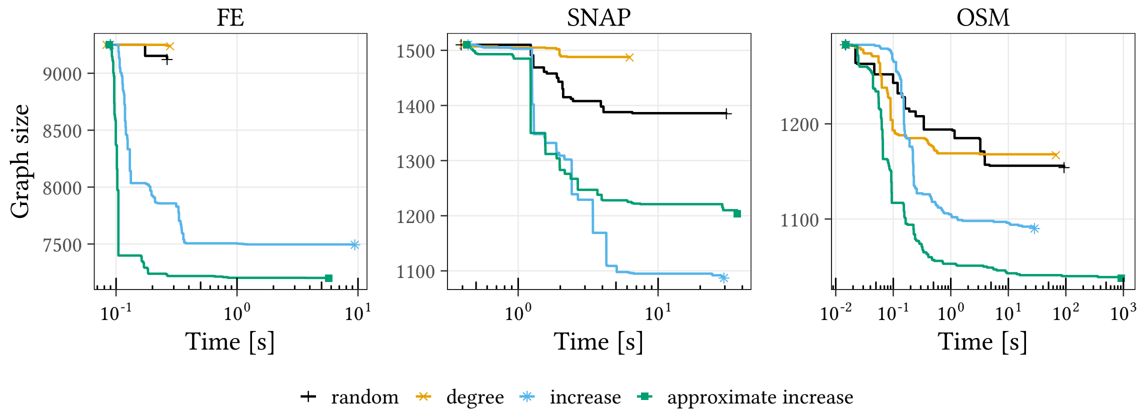



Figure A.3.: Accumulated convergence plots for configuration $C_{\text{tie break}}$ with different vertex selection strategies.

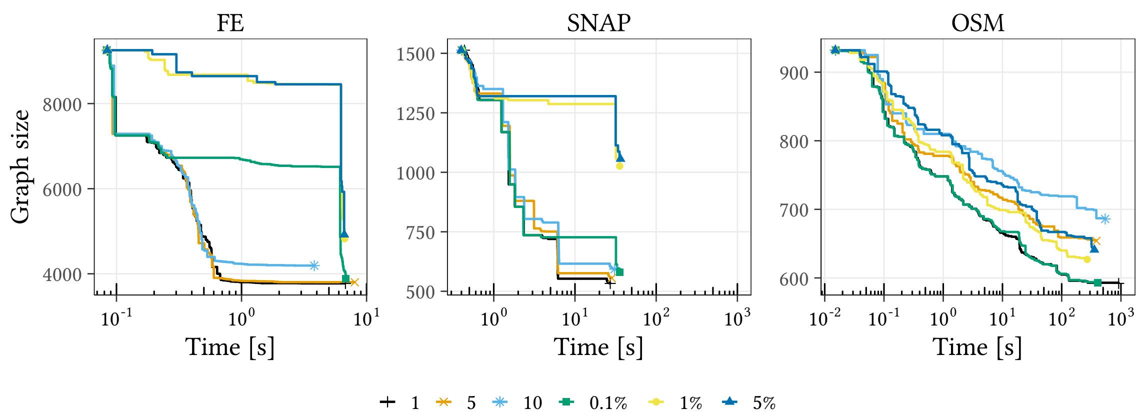



Figure A.4.: Accumulated convergence plots for the cyclic blow up algorithm with configuration $C_{\text{accept}}$ and different blow-ups per phase limit.

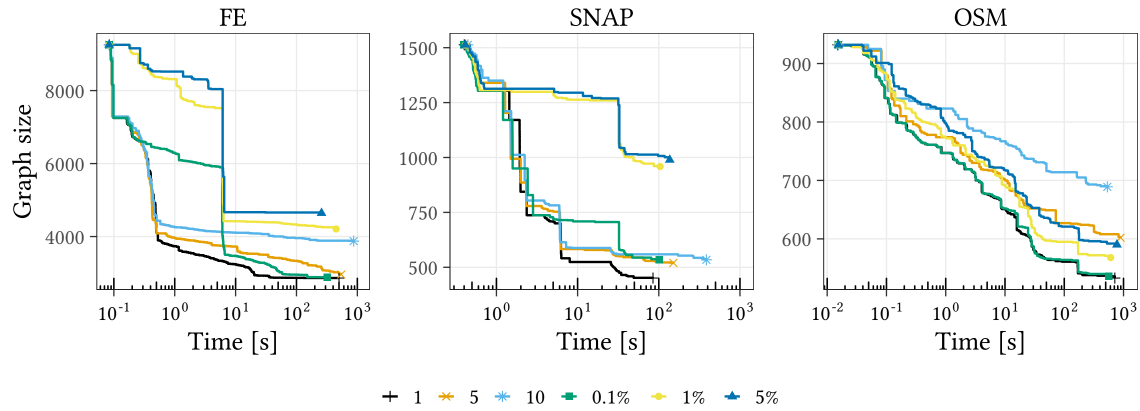



Figure A.5.: Accumulated convergence plots for the cyclic blow up algorithm with configuration $C_{\text{tabu}}$ and different blow-ups per phase limit.

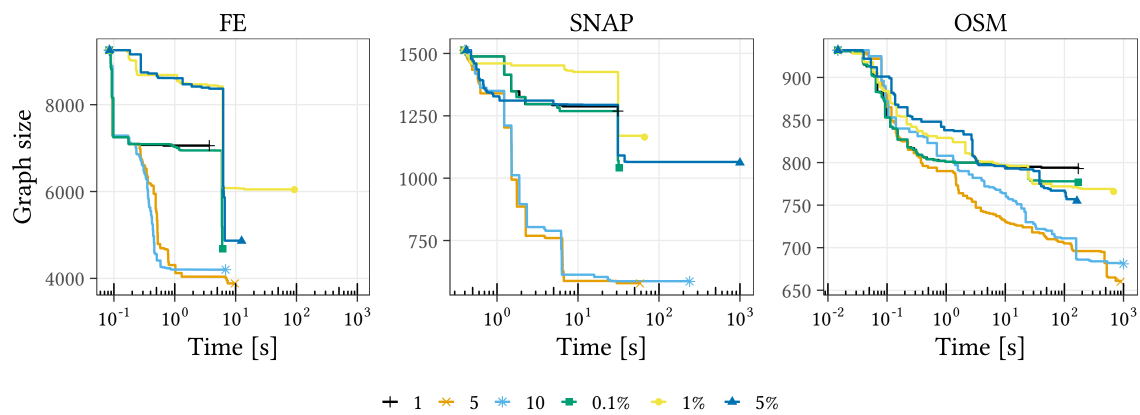



Figure A.6.: Accumulated convergence plots for the cyclic blow up algorithm with configuration $C_{\text{tie break}}$ and different blow-ups per phase limit.

## A.3. Time To Solve And Reduced Graph Size Tables

| Graph | Basic-Dense | | | Basic-Sparse | | | NonIncreasing | | | Cyclic-Fast | | | Cyclic-Strong | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ |
| fe_4elt2 | 8580 | 0.29 | - | 8578 | 0.87 | - | 562 | 0.10 | - | 0 | 0.12 | **0.13** | 0 | 0.16 | 0.17 |
| fe_body | 16107 | 0.69 | - | 15992 | 3.40 | - | 1162 | 0.16 | - | 625 | 0.44 | - | 553 | 0.94 | - |
| fe_ocean | 141283 | 1.05 | - | 0 | 5.94 | **5.99** | 138338 | 8.90 | - | 138134 | 9.61 | - | 138049 | 10.78 | - |
| fe_pwt | 34521 | 0.46 | - | 34521 | 2.70 | - | 25550 | 0.78 | - | 20241 | 1.80 | - | 14107 | 5.65 | - |
| fe_rotor | 98271 | 9.80 | - | 98271 | 24.47 | - | 91946 | 4.80 | - | 91634 | 4.82 | - | 89647 | 11.11 | - |
| fe_sphere | 15269 | 0.21 | - | 15269 | 1.47 | - | 2961 | 0.34 | - | 147 | 0.62 | **0.83** | 0 | 0.75 | 0.77 |
| fe_tooth | 10922 | 1.69 | - | 10801 | 3.79 | - | 15 | 0.41 | 0.46 | 0 | 0.30 | **0.34** | 0 | 0.28 | 0.32 |

Table A.5.: Obtained irreducible graph sizes $n$, time $t_r$ (in seconds) needed to obtain them and total solving time $t_t$ (in seconds) on FE instances. The global best solving time $t_t$ is highlighted in bold.

| Graph | Basic-Dense | | | Basic-Sparse | | | NonIncreasing | | | Cyclic-Fast | | | Cyclic-Strong | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ |
| as-skitter | 26584 | 25.82 | - | 8585 | 36.69 | - | 3426 | 4.75 | - | 2782 | 5.50 | - | 2343 | 6.80 | - |
| ca-AstroPh | 0 | 0.02 | **0.03** | 0 | 0.02 | 0.03 | 0 | 0.02 | 0.03 | 0 | 0.03 | 0.04 | 0 | 0.03 | 0.03 |
| ca-CondMat | 0 | 0.02 | 0.03 | 0 | 0.01 | 0.02 | 0 | 0.01 | **0.02** | 0 | 0.03 | 0.03 | 0 | 0.01 | 0.02 |
| ca-GrQc | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 | 0 | 0.00 | **0.00** | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.00 |
| ca-HepPh | 0 | 0.01 | 0.02 | 0 | 0.01 | 0.02 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.02 | 0 | 0.01 | 0.01 |
| ca-HepTh | 0 | 0.01 | 0.01 | 0 | 0.00 | 0.01 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 | 0 | 0.00 | 0.00 |
| email-Enron | 0 | 0.02 | **0.03** | 0 | 0.02 | 0.03 | 0 | 0.04 | 0.04 | 0 | 0.03 | 0.03 | 0 | 0.03 | 0.03 |
| email-EuAll | 0 | 0.08 | 0.17 | 0 | 0.09 | 0.16 | 0 | 0.06 | **0.08** | 0 | 0.09 | 0.13 | 0 | 0.07 | 0.10 |
| p2p-Gnutella04 | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 |
| p2p-Gnutella05 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 |
| p2p-Gnutella06 | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 |
| p2p-Gnutella08 | 0 | 0.00 | 0.00 | 0 | 0.00 | 0.01 | 0 | 0.00 | 0.00 | 0 | 0.00 | **0.00** | 0 | 0.00 | 0.00 |
| p2p-Gnutella09 | 0 | 0.00 | 0.01 | 0 | 0.01 | 0.01 | 0 | 0.00 | 0.01 | 0 | 0.00 | **0.00** | 0 | 0.00 | 0.01 |
| p2p-Gnutella24 | 0 | 0.01 | 0.02 | 0 | 0.02 | 0.03 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.02 | 0 | 0.01 | 0.01 |
| p2p-Gnutella25 | 10 | 0.01 | 0.02 | 0 | 0.01 | 0.02 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.02 | 0 | 0.02 | 0.02 |
| p2p-Gnutella30 | 0 | 0.01 | 0.02 | 0 | 0.02 | 0.03 | 0 | 0.02 | 0.02 | 0 | 0.02 | **0.02** | 0 | 0.01 | 0.02 |
| p2p-Gnutella31 | 0 | 0.04 | 0.07 | 0 | 0.04 | 0.07 | 0 | 0.03 | **0.03** | 0 | 0.05 | 0.06 | 0 | 0.04 | 0.05 |
| roadNet-CA | 234433 | 3.96 | - | 66406 | 20.51 | 437.62 | 478 | 2.14 | 5.70 | 0 | 2.42 | **3.57** | 0 | 2.59 | 3.07 |
| roadNet-PA | 133814 | 2.43 | - | 35442 | 7.73 | 23.86 | 300 | 1.05 | 2.24 | 0 | 1.19 | **1.44** | 0 | 1.14 | 1.40 |
| roadNet-TX | 153985 | 2.65 | - | 40350 | 10.49 | 24.30 | 882 | 1.23 | 3.98 | 0 | 1.32 | **1.64** | 0 | 1.34 | 1.65 |
| soc-Epinions1 | 7 | 0.05 | **0.07** | 0 | 0.06 | 0.08 | 0 | 0.08 | 0.10 | 0 | 0.07 | 0.08 | 0 | 0.07 | 0.08 |
| soc-LiveJournal1 | 60041 | 236.88 | - | 29508 | 213.74 | - | 4319 | 22.27 | - | 3530 | 24.13 | - | 1314 | 37.77 | - |
| soc-Slashdot0811 | 0 | 0.08 | 0.11 | 0 | 0.08 | 0.11 | 0 | 0.07 | **0.08** | 0 | 0.07 | 0.09 | 0 | 0.06 | 0.07 |
| soc-Slashdot0902 | 0 | 0.07 | **0.09** | 0 | 0.07 | 0.10 | 0 | 0.09 | 0.11 | 0 | 0.08 | 0.10 | 0 | 0.10 | 0.12 |
| soc-pokec-relationships | 926346 | 299.11 | - | 898779 | 1013.39 | - | 808542 | 188.57 | - | 807412 | 217.83 | - | 807395 | 388.57 | - |
| web-BerkStan | 36637 | 6.58 | - | 16661 | 8.70 | - | 1999 | 6.86 | 120.05 | 151 | 6.46 | **6.83** | 151 | 7.89 | 8.25 |
| web-Google | 2810 | 1.57 | **2.40** | 1254 | 2.42 | 3.66 | 361 | 1.75 | 2.95 | 46 | 1.88 | 2.47 | 46 | 7.97 | 9.24 |
| web-NotreDame | 13464 | 1.03 | - | 6052 | 2.03 | - | 2460 | 0.40 | - | 2061 | 0.56 | **1.60** | 117 | 2.44 | 2.57 |
| web-Stanford | 14153 | 1.81 | - | 3325 | 2.45 | - | 112 | 2.25 | 2.50 | 0 | 1.80 | **1.99** | 0 | 2.17 | 2.38 |
| wiki-Talk | 0 | 1.00 | **1.71** | 0 | 1.32 | 1.96 | 0 | 1.26 | 1.84 | 0 | 1.24 | 1.80 | 0 | 1.67 | 2.28 |
| wiki-Vote | 477 | 0.03 | 0.12 | 0 | 0.02 | 0.03 | 0 | 0.02 | **0.02** | 0 | 0.02 | 0.02 | 0 | 0.02 | 0.02 |

Table A.6.: Obtained irreducible graph sizes $n$, time $t_r$ (in seconds) needed to obtain them and total solving time $t_t$ (in seconds) on SNAP instances. The global best solving time $t_t$ is highlighted in bold.

| Graph | Basic-Dense | | | Basic-Sparse | | | NonIncreasing | | | Cyclic-Fast | | | Cyclic-Strong | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ |
| alabama-AM2 | 173 | 0.06 | 0.31 | 173 | 0.07 | 0.55 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 |
| alabama-AM3 | 1614 | 12.05 | - | 1614 | 14.37 | - | 1288 | 0.34 | - | 456 | 1.45 | **3.94** | 0 | 33.11 | 33.16 |
| district-of-columbia-AM1 | 800 | 1.22 | - | 800 | 1.28 | - | 367 | 0.03 | 39.81 | 185 | 0.41 | **0.80** | 0 | 3.65 | 3.66 |
| district-of-columbia-AM2 | 6360 | 11.86 | - | 6360 | 14.39 | - | 5606 | 0.85 | - | 1855 | 2.51 | - | 1484 | 84.91 | - |
| district-of-columbia-AM3 | 33367 | 63.23 | - | 33367 | 358.14 | - | 32320 | 33.68 | - | 28842 | 66.67 | - | 25031 | 441.44 | - |
| florida-AM2 | 41 | 0.01 | 0.01 | 41 | 0.01 | 0.01 | 0 | 0.00 | 0.00 | 0 | 0.00 | **0.00** | 0 | 0.00 | 0.00 |
| florida-AM3 | 1069 | 31.52 | 45.81 | 1069 | 35.20 | - | 814 | 0.13 | 3.85 | 661 | 0.44 | **2.93** | 267 | 42.26 | 45.05 |
| georgia-AM3 | 861 | 8.99 | 892.17 | 861 | 10.14 | - | 796 | 0.08 | 25.97 | 587 | 0.69 | **10.35** | 425 | 12.84 | 32.53 |
| greenland-AM3 | 3942 | 3.81 | - | 3942 | 24.77 | - | 3953 | 3.94 | - | 3339 | 10.27 | - | 3339 | 54.44 | - |
| hawaii-AM2 | 428 | 2.08 | 4.27 | 428 | 2.15 | 10.22 | 262 | 0.07 | 0.18 | 0 | 0.09 | **0.09** | 0 | 0.10 | 0.10 |
| hawaii-AM3 | 24436 | 70.38 | - | 24436 | 743.04 | - | 24184 | 98.22 | - | 22997 | 118.52 | - | 21087 | 632.02 | - |
| idaho-AM3 | 3208 | 3.17 | - | 3208 | 29.91 | - | 3204 | 6.96 | - | 3160 | 8.74 | - | 2909 | 33.77 | - |
| kansas-AM3 | 1605 | 2.46 | - | 1605 | 4.81 | - | 1550 | 0.49 | - | 903 | 2.46 | **430.93** | 860 | 41.61 | 489.15 |
| kentucky-AM2 | 442 | 2.05 | 11.85 | 442 | 2.19 | 67.28 | 183 | 0.20 | 0.39 | 0 | 0.22 | **0.23** | 0 | 0.41 | 0.42 |
| kentucky-AM3 | 16871 | 109.47 | - | 16871 | 3344.67 | - | 16807 | 237.86 | - | 15947 | 298.49 | - | 15684 | 705.46 | - |
| louisiana-AM3 | 382 | 4.56 | 6.55 | 382 | 5.04 | 25.22 | 349 | 0.03 | 0.82 | 0 | 0.07 | **0.07** | 0 | 0.16 | 0.16 |
| maryland-AM3 | 187 | 7.59 | 8.49 | 187 | 8.65 | 10.73 | 335 | 0.03 | 0.19 | 0 | 0.11 | **0.11** | 0 | 0.15 | 0.15 |
| massachusetts-AM2 | 196 | 0.04 | 0.36 | 196 | 0.04 | 0.58 | 193 | 0.02 | 0.07 | 0 | 0.06 | **0.06** | 0 | 0.07 | 0.07 |
| massachusetts-AM3 | 2008 | 9.42 | - | 2008 | 12.62 | - | 1928 | 0.36 | - | 1636 | 1.08 | - | 1632 | 31.83 | - |
| mexico-AM3 | 620 | 25.29 | 80.23 | 620 | 27.52 | 991.99 | 514 | 0.03 | **1.47** | 483 | 0.28 | 1.50 | 0 | 21.03 | 21.30 |
| new-hampshire-AM3 | 247 | 4.99 | 6.19 | 247 | 5.69 | 15.89 | 164 | 0.02 | 0.17 | 0 | 0.07 | **0.07** | 0 | 0.09 | 0.09 |
| north-carolina-AM3 | 1178 | 0.69 | - | 1178 | 1.22 | - | 1146 | 0.25 | - | 1144 | 0.43 | - | 700 | 47.38 | 379.088 |
| oregon-AM2 | 35 | 0.04 | 0.05 | 35 | 0.05 | 0.05 | 0 | 0.01 | **0.01** | 0 | 0.02 | 0.02 | 0 | 0.01 | 0.01 |
| oregon-AM3 | 3670 | 9.95 | - | 3670 | 34.95 | - | 3584 | 3.92 | - | 3417 | 6.21 | - | 2721 | 38.72 | - |
| pennsylvania-AM3 | 315 | 16.69 | 20.71 | 315 | 19.39 | 113.87 | 317 | 0.03 | 0.39 | 0 | 0.07 | **0.07** | 0 | 0.12 | 0.12 |
| rhode-island-AM2 | 1103 | 0.55 | - | 1103 | 0.68 | - | 845 | 0.17 | 163.07 | 0 | 0.53 | **0.53** | 0 | 4.57 | 4.58 |
| rhode-island-AM3 | 13031 | 7.75 | - | 13031 | 193.76 | - | 12934 | 26.54 | - | 12653 | 29.75 | - | 12653 | 59.69 | - |
| utah-AM3 | 568 | 8.21 | 51.91 | 568 | 8.97 | 276.27 | 396 | 0.03 | 0.87 | 0 | 0.09 | **0.09** | 0 | 0.40 | 0.41 |
| vermont-AM3 | 2630 | 4.79 | - | 2630 | 9.82 | - | 2289 | 0.97 | - | 2069 | 1.37 | - | 2045 | 55.28 | - |
| virginia-AM2 | 237 | 0.13 | 0.61 | 237 | 0.12 | 0.99 | 0 | 0.03 | **0.03** | 0 | 0.03 | 0.03 | 0 | 0.03 | 0.03 |
| virginia-AM3 | 3867 | 34.13 | - | 3867 | 39.74 | - | 3738 | 0.40 | - | 2827 | 1.28 | - | 2547 | 81.67 | - |
| washington-AM2 | 382 | 0.24 | 5.31 | 382 | 0.18 | 8.58 | 171 | 0.05 | 0.37 | 0 | 0.06 | **0.06** | 0 | 0.07 | 0.07 |
| washington-AM3 | 8030 | 50.21 | - | 8030 | 67.00 | - | 7649 | 2.19 | - | 6895 | 3.12 | - | 6159 | 73.52 | - |
| west-virginia-AM3 | 991 | 10.69 | - | 991 | 12.13 | - | 970 | 0.08 | 238.39 | 890 | 0.33 | **155.49** | 881 | 38.73 | 241.68 |

Table A.7.: Obtained irreducible graph sizes $n$, time $t_r$ (in seconds) needed to obtain them and total solving time $t_t$ (in seconds) on OSM instances. The global best solving time $t_t$ is highlighted in bold.

| Graph | Basic-Dense | | | Basic-Sparse | | | NonIncreasing | | | Cyclic-Fast | | | Cyclic-Strong | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ | $n$ | $t_r$ | $t_t$ |
| beethoven | 1254 | 0.02 | 7.86 | 427 | 0.02 | 0.08 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 | 0 | 0.01 | 0.01 |
| blob | 5746 | 0.08 | - | 1464 | 0.06 | 0.20 | 0 | 0.03 | **0.03** | 0 | 0.03 | 0.04 | 0 | 0.03 | 0.03 |
| buddha | 380315 | 5.56 | - | 107265 | 26.19 | 67.85 | 86 | 1.83 | 2.74 | 0 | 1.87 | **2.26** | 0 | 1.91 | 2.39 |
| bunny | 24580 | 0.34 | - | 3290 | 0.56 | 0.89 | 0 | 0.12 | **0.14** | 0 | 0.13 | 0.16 | 0 | 0.15 | 0.18 |
| cow | 1916 | 0.02 | - | 513 | 0.02 | 0.06 | 0 | 0.01 | 0.01 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 |
| dragon | 51885 | 0.89 | - | 12893 | 1.34 | 3.83 | 0 | 0.18 | **0.21** | 0 | 0.19 | 0.23 | 0 | 0.21 | 0.25 |
| dragonsub | 218779 | 2.60 | - | 19470 | 4.15 | 5.66 | 506 | 1.03 | 2.08 | 0 | 1.13 | **1.36** | 0 | 1.07 | 1.28 |
| ecat | 239787 | 4.07 | - | 26270 | 10.09 | 12.93 | 274 | 2.12 | 3.16 | 0 | 2.12 | **2.51** | 0 | 2.14 | 2.56 |
| face | 7588 | 0.09 | - | 1540 | 0.10 | 0.21 | 0 | 0.03 | 0.04 | 0 | 0.03 | **0.03** | 0 | 0.03 | 0.04 |
| fandisk | 2851 | 0.05 | - | 336 | 0.03 | 0.07 | 51 | 0.02 | 0.03 | 0 | 0.02 | **0.02** | 0 | 0.02 | 0.02 |
| feline | 14817 | 0.20 | - | 2743 | 0.25 | 0.47 | 0 | 0.08 | 0.09 | 0 | 0.08 | **0.09** | 0 | 0.08 | 0.09 |
| gameguy | 13959 | 0.17 | - | 312 | 0.10 | 0.12 | 0 | 0.06 | 0.07 | 0 | 0.06 | **0.07** | 0 | 0.06 | 0.07 |
| gargoyle | 6512 | 0.15 | - | 1819 | 0.14 | 0.36 | 0 | 0.03 | 0.03 | 0 | 0.03 | **0.03** | 0 | 0.03 | 0.03 |
| turtle | 91624 | 1.17 | - | 16095 | 1.92 | 4.98 | 186 | 0.42 | 0.65 | 0 | 0.41 | **0.49** | 0 | 0.47 | 0.56 |
| venus | 1898 | 0.02 | - | 175 | 0.01 | 0.02 | 0 | 0.01 | 0.01 | 0 | 0.01 | **0.01** | 0 | 0.01 | 0.01 |

Table A.8.: Obtained irreducible graph sizes $n$, time $t_r$ (in seconds) needed to obtain them and total solving time $t_t$ (in seconds) on mesh instances. The global best solving time $t_t$ is highlighted in bold.

## A.4. Best Solution Tables

| Graph | DynWVC1 | | DynWVC2 | | HILS | | Cyclic-Fast | | Cyclic-Strong | | Non-Increasing | | Basic-Sparse | | Basic-Dense | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ |
| fe_4elt2 | 961.12 | 427755 | 974.87 | 427755 | 759.23 | 427646 | 0.11 | **428029** | 0.11 | **428029** | 0.18 | 428016 | 0.66 | 420477 | 0.24 | 421235 |
| fe_body | 504.31 | 1678616 | 499.03 | 1678496 | 806.46 | 1678708 | 0.51 | **1680182** | 0.86 | 1680117 | 0.27 | 1680133 | 858.22 | 1194619 | 26.78 | 1127790 |
| fe_ocean | 983.53 | 7222521 | 379.75 | 7220128 | 999.57 | 7069279 | 18.85 | 6591832 | 19.04 | 6591537 | 18.85 | 6597698 | 4.91 | **7248581** | 3.35 | 6604880 |
| fe_pwt | 814.23 | 1176721 | 320.05 | **1176784** | 932.43 | 1175754 | 3.03 | 1162232 | 5.45 | 888959 | 1.57 | 1151777 | 3.02 | 1132622 | 0.79 | 1132622 |
| fe_rotor | 961.76 | **2659653** | 874.68 | 2659473 | 973.92 | 2650132 | 13.95 | 2531152 | 20.55 | 2538117 | 13.56 | 2532168 | 20.76 | 2496992 | 10.12 | 2496992 |
| fe_sphere | 875.87 | 616978 | 872.36 | 616978 | 843.67 | 616528 | 0.63 | **617816** | 0.67 | **617816** | 0.46 | 617585 | 1.10 | 600936 | 0.45 | 600164 |
| fe_tooth | 353.21 | 3031269 | 619.96 | 3031385 | 994.97 | 3032819 | 0.26 | **3033298** | 0.26 | **3033298** | 0.27 | **3033298** | 13.02 | 2694792 | 1.55 | 2677851 |

Table A.9.: Best solution found by each algorithm on FE instances and time (in seconds) required to compute it. The global best solution is highlighted in bold. Rows are highlighted in gray if an exact solver is able to solve the corresponding instances.

| Graph | DynWVC1 | | DynWVC2 | | HILS | | Cyclic-Fast | | Cyclic-Strong | | Non-Increasing | | Basic-Sparse | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ |
| as-skitter | 989.05 | 123613404 | 383.97 | 123273938 | 999.32 | 122658804 | 346.69 | 124137148 | 354.71 | **124137365** | 431.90 | 124136621 | 801.99 | 124025255 |
| ca-AstroPh | 32.46 | 797475 | 125.05 | 797480 | 13.47 | **797510** | 0.02 | **797510** | 0.02 | **797510** | 0.02 | **797510** | 0.02 | **797510** |
| ca-CondMat | 114.85 | 1147814 | 27.75 | 1147845 | 50.90 | **1147950** | 0.01 | **1147950** | 0.01 | **1147950** | 0.01 | **1147950** | 0.02 | **1147950** |
| ca-GrQc | 4.87 | **286489** | 1.93 | **286489** | 0.34 | **286489** | 0.00 | **286489** | 0.00 | **286489** | 0.00 | **286489** | 0.00 | **286489** |
| ca-HepPh | 13.21 | 581014 | 17.34 | 581028 | 7.73 | **581039** | 0.01 | **581039** | 0.01 | **581039** | 0.01 | **581039** | 0.01 | **581039** |
| ca-HepTh | 6.57 | 561982 | 5.30 | 561974 | 4.68 | **562004** | 0.00 | **562004** | 0.00 | **562004** | 0.00 | **562004** | 0.00 | **562004** |
| email-Enron | 454.49 | 2464887 | 594.93 | 2464890 | 71.07 | 2464922 | 0.02 | **2464935** | 0.03 | **2464935** | 0.02 | **2464935** | 0.02 | **2464935** |
| email-EuAll | 134.83 | **25286322** | 132.62 | **25286322** | 338.14 | **25286322** | 0.07 | **25286322** | 0.07 | **25286322** | 0.06 | **25286322** | 0.09 | **25286322** |
| p2p-Gnutella04 | 1.46 | 679105 | 2.34 | **679111** | 94.12 | **679111** | 0.01 | **679111** | 0.01 | **679111** | 0.01 | **679111** | 0.01 | **679111** |
| p2p-Gnutella05 | 1.15 | 554926 | 3.55 | 554931 | 135.17 | **554943** | 0.01 | **554943** | 0.01 | **554943** | 0.01 | **554943** | 0.01 | **554943** |
| p2p-Gnutella06 | 525.35 | 548611 | 186.97 | 548611 | 1.29 | **548612** | 0.01 | **548612** | 0.01 | **548612** | 0.01 | **548612** | 0.01 | **548612** |
| p2p-Gnutella08 | 0.15 | 434575 | 0.18 | **434577** | 0.12 | **434577** | 0.00 | **434577** | 0.00 | **434577** | 0.00 | **434577** | 0.00 | **434577** |
| p2p-Gnutella09 | 0.39 | **568439** | 0.28 | **568439** | 0.09 | **568439** | 0.00 | **568439** | 0.00 | **568439** | 0.00 | **568439** | 0.00 | **568439** |
| p2p-Gnutella24 | 8.01 | **1984567** | 5.51 | **1984567** | 3.17 | **1984567** | 0.01 | **1984567** | 0.01 | **1984567** | 0.01 | **1984567** | 0.01 | **1984567** |
| p2p-Gnutella25 | 2.66 | **1701967** | 2.20 | **1701967** | 1.17 | **1701967** | 0.01 | **1701967** | 0.01 | **1701967** | 0.01 | **1701967** | 0.01 | **1701967** |
| p2p-Gnutella30 | 8.83 | 2787903 | 132.71 | 2787899 | 15.14 | **2787907** | 0.01 | **2787907** | 0.01 | **2787907** | 0.02 | **2787907** | 0.02 | **2787907** |
| p2p-Gnutella31 | 70.88 | 4776960 | 47.97 | 4776961 | 115.01 | **4776986** | 0.02 | **4776986** | 0.02 | **4776986** | 0.03 | **4776986** | 0.03 | **4776986** |
| roadNet-CA | 999.98 | 109586054 | 999.90 | 109582579 | 1000.00 | 106584645 | 1.94 | **111360828** | 1.86 | **111360828** | 4.09 | **111360828** | 437.34 | **111360828** |
| roadNet-PA | 511.59 | 60990177 | 469.18 | 60990177 | 999.94 | 60037011 | 0.96 | **61731589** | 1.04 | **61731589** | 1.83 | **61731589** | 16.49 | **61731589** |
| roadNet-TX | 789.43 | 77672388 | 694.33 | 77672388 | 999.97 | 76347666 | 1.29 | **78599946** | 1.29 | **78599946** | 3.42 | **78599946** | 22.78 | **78599946** |
| soc-Epinions1 | 290.84 | 5690651 | 272.56 | 5690773 | 253.10 | 5690874 | 0.08 | **5690970** | 0.08 | **5690970** | 0.08 | **5690970** | 0.07 | **5690970** |
| soc-LiveJournal1 | 999.99 | 279150686 | 999.99 | 279231875 | 1000.00 | 255079926 | 51.33 | 284036222 | 44.19 | **284036239** | 39.36 | 283970295 | 231.80 | 284010263 |
| soc-Slashdot0811 | 238.18 | 5660385 | 880.68 | 5660555 | 446.95 | 5660787 | 0.09 | **5660899** | 0.08 | **5660899** | 0.08 | **5660899** | 0.08 | **5660899** |
| soc-Slashdot0902 | 270.85 | 5971308 | 435.90 | 5971476 | 604.07 | 5971664 | 0.11 | **5971849** | 0.11 | **5971849** | 0.12 | **5971849** | 0.11 | **5971849** |
| soc-pokec-relationships | 999.85 | **83223668** | 999.13 | 83155217 | 1000.00 | 82021946 | 254.59 | 76075111 | 488.31 | 76075700 | 228.07 | 76063476 | 0.00 | 0 |
| web-BerkStan | 194.20 | 43640833 | 164.10 | 43637382 | 998.73 | 43424373 | 6.74 | **43907482** | 8.05 | **43907482** | 16.01 | **43907482** | 50.88 | 43742339 |
| web-Google | 349.08 | 56209005 | 324.65 | 56206250 | 995.92 | 56008278 | 1.72 | **56326504** | 6.44 | **56326504** | 2.17 | **56326504** | 2.86 | **56326504** |
| web-NotreDame | 949.84 | 26010791 | 905.72 | 26009287 | 997.00 | 26002793 | 1.60 | **26016941** | 2.74 | **26016941** | 1.36 | **26016941** | 99.49 | **26016941** |
| web-Stanford | 943.85 | 17748798 | 671.32 | 17741043 | 999.50 | 17709827 | 1.68 | **17792930** | 1.86 | **17792930** | 1.71 | **17792930** | 2.51 | 17792824 |
| wiki-Talk | 951.51 | 235836837 | 972.93 | 235836913 | 999.69 | 235818823 | 1.29 | **235837346** | 1.29 | **235837346** | 1.31 | **235837346** | 1.30 | **235837346** |
| wiki-Vote | 188.76 | 500075 | 0.32 | **500079** | 10.34 | **500079** | 0.02 | **500079** | 0.02 | **500079** | 0.02 | **500079** | 0.02 | **500079** |

Table A.10.: Best solution found by each algorithm on SNAP instances and time (in seconds) required to compute it. The global best solution is highlighted in bold. Rows are highlighted in gray if an exact solver is able to solve the corresponding instances.

| Graph | DynWVC1 | | DynWVC2 | | HILS | | Cyclic-Fast | | Cyclic-Strong | | Non-Increasing | | Basic-Sparse | | Basic-Dense | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ |
| alabama-AM2 | 0.18 | 174252 | 0.24 | 174269 | 0.03 | **174309** | 0.01 | **174309** | 0.01 | **174309** | 0.01 | **174309** | 0.28 | **174309** | 0.28 | **174309** |
| alabama-AM3 | 725.34 | 185518 | 199.94 | 185655 | 0.58 | **185744** | 1.76 | **185744** | 32.42 | **185744** | 0.60 | **185744** | 71.87 | 185707 | 22.23 | 185707 |
| district-of-columbia-AM1 | 23.96 | **196475** | 28.42 | **196475** | 0.14 | **196475** | 0.32 | **196475** | 3.52 | **196475** | 0.06 | **196475** | 2.46 | **196475** | 2.01 | **196475** |
| district-of-columbia-AM2 | 159.08 | 208989 | 915.18 | 208977 | 400.69 | **209132** | 4.21 | **209132** | 84.21 | 209131 | 686.26 | 174114 | 214.18 | 148513 | 23.26 | 148513 |
| district-of-columbia-AM3 | 461.10 | 224760 | 313.17 | 223955 | 849.37 | **227613** | 904.91 | 142454 | 804.79 | 156967 | 168.55 | 120366 | 673.89 | 92804 | 65.42 | 92804 |
| florida-AM2 | 0.18 | **230595** | 0.53 | **230595** | 0.04 | **230595** | 0.00 | **230595** | 0.00 | **230595** | 0.00 | **230595** | 0.01 | **230595** | 0.01 | **230595** |
| florida-AM3 | 425.87 | 237229 | 862.04 | 237120 | 3.98 | **237333** | 1.57 | **237333** | 40.97 | **237333** | 2.08 | **237333** | 305.20 | 226767 | 42.70 | **237333** |
| georgia-AM3 | 0.42 | **222652** | 1.31 | **222652** | 0.04 | **222652** | 0.98 | **222652** | 12.97 | **222652** | 14.56 | **222652** | 861.05 | **222652** | 842.78 | **222652** |
| greenland-AM3 | 58.88 | 14007 | 640.46 | 14010 | 1.18 | 14011 | 10.95 | 14011 | 58.24 | 14008 | 5.06 | **14012** | 35.22 | 13829 | 15.89 | 13829 |
| hawaii-AM2 | 1.89 | 125270 | 1.63 | 125270 | 0.20 | **125284** | 0.09 | **125284** | 0.10 | **125284** | 0.13 | **125284** | 8.44 | **125284** | 4.05 | **125284** |
| hawaii-AM3 | 406.57 | 140656 | 887.44 | 140595 | 213.32 | **141035** | 152.38 | 116202 | 681.39 | 121222 | 155.21 | 107879 | 771.31 | 96774 | 204.48 | 106251 |
| idaho-AM3 | 79.67 | **77145** | 58.83 | **77145** | 0.78 | **77145** | 11.95 | 77141 | 40.71 | 77144 | 8.89 | 77144 | 42.51 | 76991 | 455.18 | 77010 |
| kansas-AM3 | 333.60 | **87976** | 276.26 | **87976** | 0.55 | **87976** | 2.25 | **87976** | 110.41 | **87976** | 337.83 | **87976** | 14.95 | 87955 | 12.67 | 87955 |
| kentucky-AM2 | 3.23 | **97397** | 2.92 | **97397** | 0.26 | **97397** | 0.23 | **97397** | 0.44 | **97397** | 0.26 | **97397** | 36.56 | **97397** | 11.72 | **97397** |
| kentucky-AM3 | 951.91 | 100476 | 96.83 | 100455 | 515.99 | 100507 | 354.45 | **100510** | 776.69 | **100510** | 305.01 | 100497 | 0.00 | 0 | 831.07 | 100311 |
| louisiana-AM3 | 8.63 | **60024** | 0.18 | 60002 | 0.01 | **60024** | 0.05 | **60024** | 0.11 | **60024** | 0.15 | **60024** | 18.50 | **60024** | 5.96 | **60024** |
| maryland-AM3 | 0.79 | **45496** | 0.59 | **45496** | 0.01 | **45496** | 0.11 | **45496** | 0.15 | **45496** | 0.14 | **45496** | 10.01 | **45496** | 8.46 | **45496** |
| massachusetts-AM2 | 0.25 | **140095** | 0.74 | **140095** | 0.01 | **140095** | 0.04 | **140095** | 0.05 | **140095** | 0.03 | **140095** | 0.32 | **140095** | 0.31 | **140095** |
| massachusetts-AM3 | 980.11 | 145852 | 270.28 | 145862 | 0.77 | **145866** | 1.39 | **145866** | 31.04 | **145866** | 0.76 | **145866** | 22.52 | 145819 | 19.32 | 145819 |
| mexico-AM3 | 0.71 | **97663** | 2.28 | **97663** | 0.02 | **97663** | 0.96 | **97663** | 21.19 | **97663** | 0.67 | **97663** | 259.31 | **97663** | 32.63 | **97663** |
| new-hampshire-AM3 | 0.08 | **116060** | 1.63 | **116060** | 0.03 | **116060** | 0.05 | **116060** | 0.08 | **116060** | 0.06 | **116060** | 7.32 | **116060** | 6.09 | **116060** |
| north-carolina-AM3 | 0.58 | 49694 | 114.45 | **49720** | 0.03 | **49720** | 0.74 | **49720** | 45.82 | **49720** | 0.47 | **49720** | 11.20 | 49563 | 10.67 | 49563 |
| oregon-AM2 | 0.62 | **165047** | 0.37 | **165047** | 0.02 | **165047** | 0.01 | **165047** | 0.01 | **165047** | 0.01 | **165047** | 0.04 | **165047** | 0.03 | **165047** |
| oregon-AM3 | 174.64 | 175059 | 511.10 | 175067 | 4.65 | **175078** | 9.50 | **175078** | 39.78 | 175077 | 21.29 | **175078** | 418.10 | 164941 | 282.50 | 174931 |
| pennsylvania-AM3 | 0.06 | **143870** | 0.14 | **143870** | 0.02 | **143870** | 0.07 | **143870** | 0.12 | **143870** | 0.16 | **143870** | 33.01 | **143870** | 20.61 | **143870** |
| rhode-island-AM2 | 7.75 | 184537 | 13.90 | 184576 | 0.24 | **184596** | 0.41 | **184596** | 4.37 | **184596** | 0.27 | **184596** | 14.50 | **184596** | 9.87 | **184596** |
| rhode-island-AM3 | 230.53 | 201470 | 711.97 | 201359 | 30.15 | **201758** | 44.88 | 167162 | 82.02 | 167162 | 45.46 | 166103 | 196.07 | 163150 | 13.44 | 163150 |
| utah-AM3 | 215.88 | 98802 | 136.90 | **98847** | 0.07 | **98847** | 0.09 | **98847** | 0.27 | **98847** | 0.44 | **98847** | 29.25 | **98847** | 49.25 | **98847** |
| vermont-AM3 | 28.77 | 63234 | 768.43 | 63248 | 979.14 | 63310 | 145.39 | **63312** | 448.54 | **63312** | 217.67 | **63312** | 357.47 | 55577 | 73.99 | 55584 |
| virginia-AM2 | 0.53 | 295758 | 20.50 | 295638 | 0.07 | **295867** | 0.02 | **295867** | 0.02 | **295867** | 0.02 | **295867** | 0.55 | **295867** | 0.56 | **295867** |
| virginia-AM3 | 754.86 | 307782 | 809.24 | 307907 | 2.52 | **308305** | 34.42 | **308305** | 200.13 | **308305** | 49.42 | **308305** | 968.54 | 247790 | 109.05 | 307741 |
| washington-AM2 | 1.24 | **305619** | 13.35 | **305619** | 0.25 | **305619** | 0.06 | **305619** | 0.07 | **305619** | 0.08 | **305619** | 1.53 | **305619** | 1.60 | **305619** |
| washington-AM3 | 37.94 | 313689 | 383.62 | 313844 | 10.17 | **314288** | 3.60 | 284684 | 72.84 | 288116 | 4.56 | 282020 | 469.20 | 272404 | 228.47 | 272404 |
| west-virginia-AM3 | 2.75 | **47927** | 2.84 | **47927** | 0.07 | **47927** | 2.88 | **47927** | 41.73 | **47927** | 2.60 | **47927** | 831.85 | **47927** | 22.09 | **47927** |

Table A.11.: Best solution found by each algorithm on OSM instances and time (in seconds) required to compute it. The global best solution is highlighted in bold. Rows are highlighted in gray if an exact solver is able to solve the corresponding instances.

| Graph | DynWVC1 | | DynWVC2 | | HILS | | Cyclic-Fast | | Cyclic-Strong | | Non-Increasing | | Basic-Sparse | | Basic-Dense | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ | $t_{max}$ | $w_{max}$ |
| beethoven | 8.86 | 238726 | 8.79 | 238726 | 462.31 | 238746 | 0.00 | **238794** | 0.00 | **238794** | 0.00 | **238794** | 0.05 | **238794** | 5.37 | **238794** |
| blob | 39.91 | 854843 | 40.00 | 854843 | 351.91 | 855004 | 0.02 | **855547** | 0.02 | **855547** | 0.02 | **855547** | 0.19 | **855547** | 0.14 | 616613 |
| buddha | 879.42 | 56757052 | 797.35 | 56757052 | 999.94 | 55490134 | 1.75 | **57555880** | 1.77 | **57555880** | 2.24 | **57555880** | 60.29 | **57555880** | 742.47 | 41423703 |
| bunny | 702.13 | 3683000 | 695.55 | 3683000 | 964.60 | 3681696 | 0.11 | **3686960** | 0.13 | **3686960** | 0.11 | **3686960** | 0.57 | **3686960** | 44.84 | 3652620 |
| cow | 62.04 | 269340 | 61.40 | 269340 | 935.58 | 269464 | 0.01 | **269543** | 0.01 | **269543** | 0.01 | **269543** | 0.06 | **269543** | 0.22 | 269206 |
| dragon | 970.34 | 7943911 | 981.51 | 7944042 | 996.01 | 7940422 | 0.21 | **7956530** | 0.22 | **7956530** | 0.22 | **7956530** | 3.82 | **7956530** | 926.01 | 5951672 |
| dragonsub | 323.07 | 31762035 | 379.11 | 31762035 | 999.54 | 31304363 | 1.10 | **32213898** | 1.11 | **32213898** | 1.88 | **32213898** | 5.11 | **32213898** | 936.72 | 22327211 |
| ecat | 565.03 | 36129804 | 542.87 | 36129804 | 999.91 | 35512644 | 2.19 | **36650298** | 2.29 | **36650298** | 2.44 | **36650298** | 12.26 | **36650298** | 899.38 | 25773421 |
| face | 87.05 | 1218510 | 86.38 | 1218510 | 228.77 | 1218565 | 0.03 | **1219418** | 0.03 | **1219418** | 0.03 | **1219418** | 0.21 | **1219418** | 22.99 | 943327 |
| fandisk | 8.26 | 462950 | 8.42 | 462950 | 232.96 | 463090 | 0.01 | **463288** | 0.01 | **463288** | 0.01 | **463288** | 0.04 | **463288** | 5.53 | 462200 |
| feline | 730.80 | 2204925 | 734.34 | 2204925 | 640.98 | 2204911 | 0.09 | **2207219** | 0.08 | **2207219** | 0.09 | **2207219** | 0.51 | **2207219** | 270.27 | 1713601 |
| gameguy | 519.12 | 2323941 | 525.93 | 2323941 | 736.64 | 2322824 | 0.05 | **2325878** | 0.05 | **2325878** | 0.05 | **2325878** | 0.11 | **2325878** | 177.00 | 1904924 |
| gargoyle | 29.25 | 1058496 | 29.11 | 1058496 | 724.41 | 1058652 | 0.03 | **1059559** | 0.03 | **1059559** | 0.03 | **1059559** | 0.28 | **1059559** | 229.56 | 875811 |
| turtle | 982.00 | 14215429 | 976.57 | 14213516 | 999.68 | 14151616 | 0.42 | **14263005** | 0.43 | **14263005** | 0.56 | **14263005** | 4.58 | **14263005** | 407.58 | 10233723 |
| venus | 559.29 | 305571 | 556.38 | 305571 | 130.83 | 305724 | 0.01 | **305749** | 0.01 | **305749** | 0.01 | **305749** | 0.02 | **305749** | 0.08 | 304452 |

Table A.12.: Best solution found by each algorithm on mesh instances and time (in seconds) required to compute it. The global best solution is highlighted in bold. Rows are highlighted in gray if an exact solver is able to solve the corresponding instances.

## A.5. Solution Quality Convergence Plots

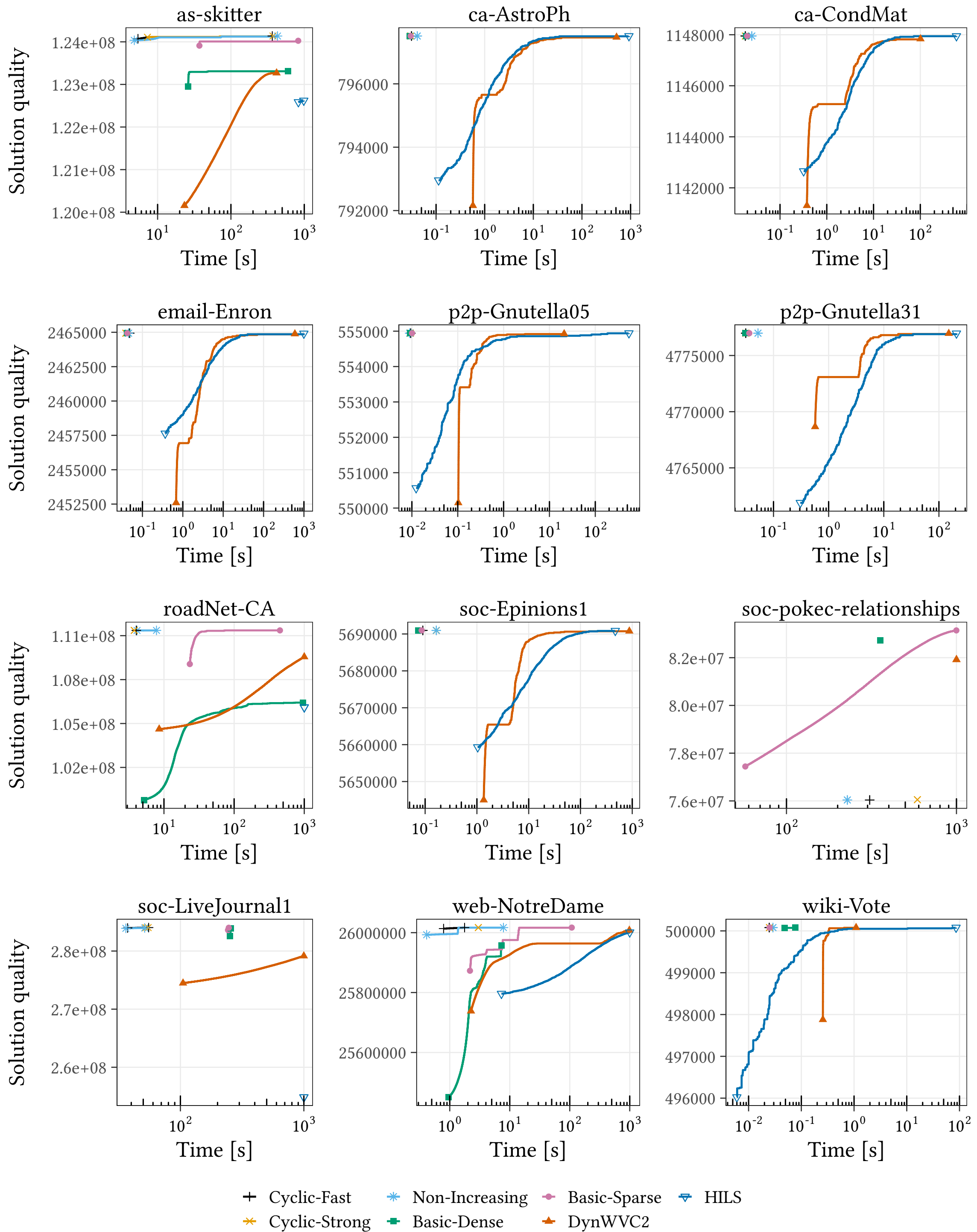



Figure A.7.: Solution quality over time for 12 SNAP instances.

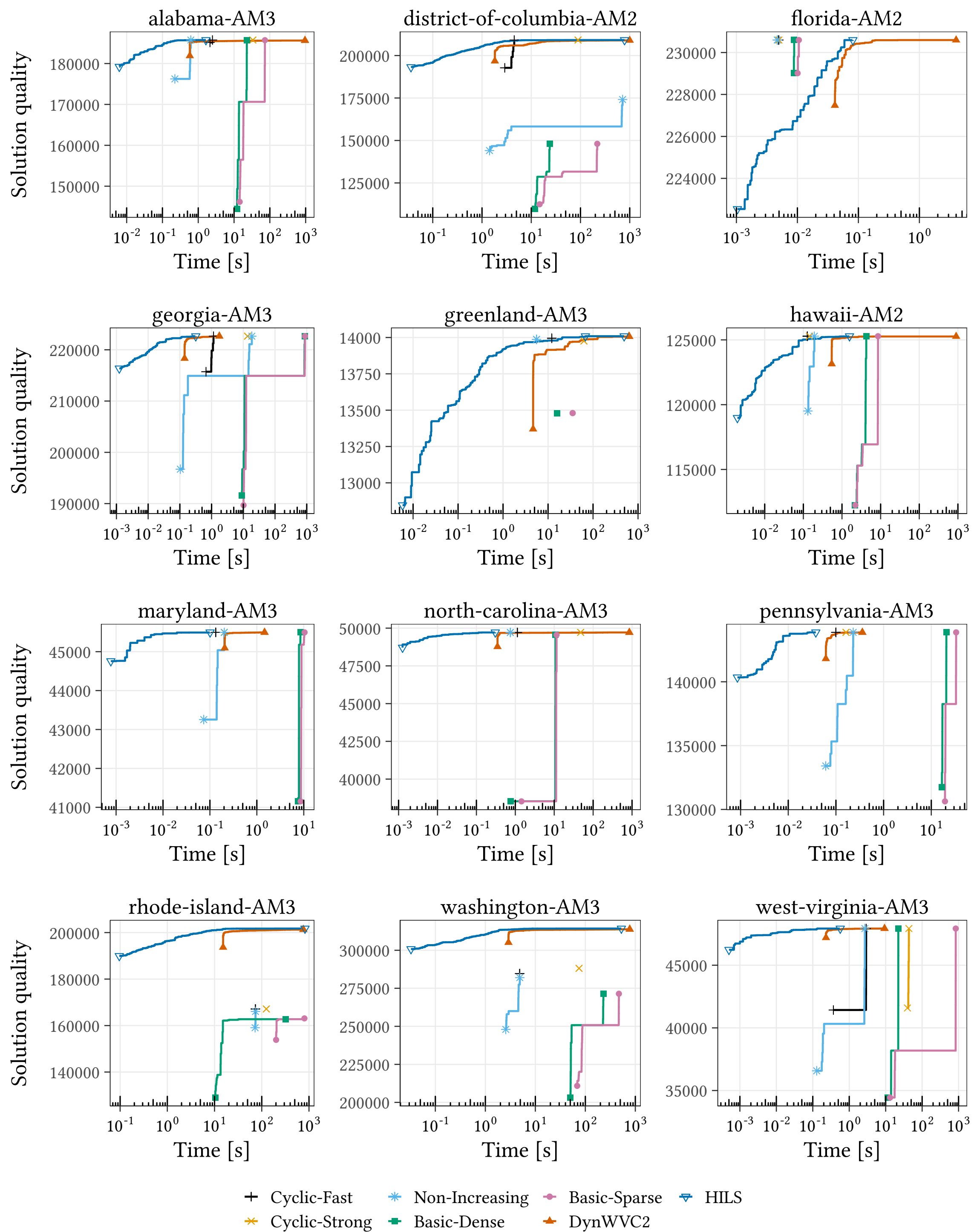



Figure A.8.: Solution quality over time for 12 OSM instances.

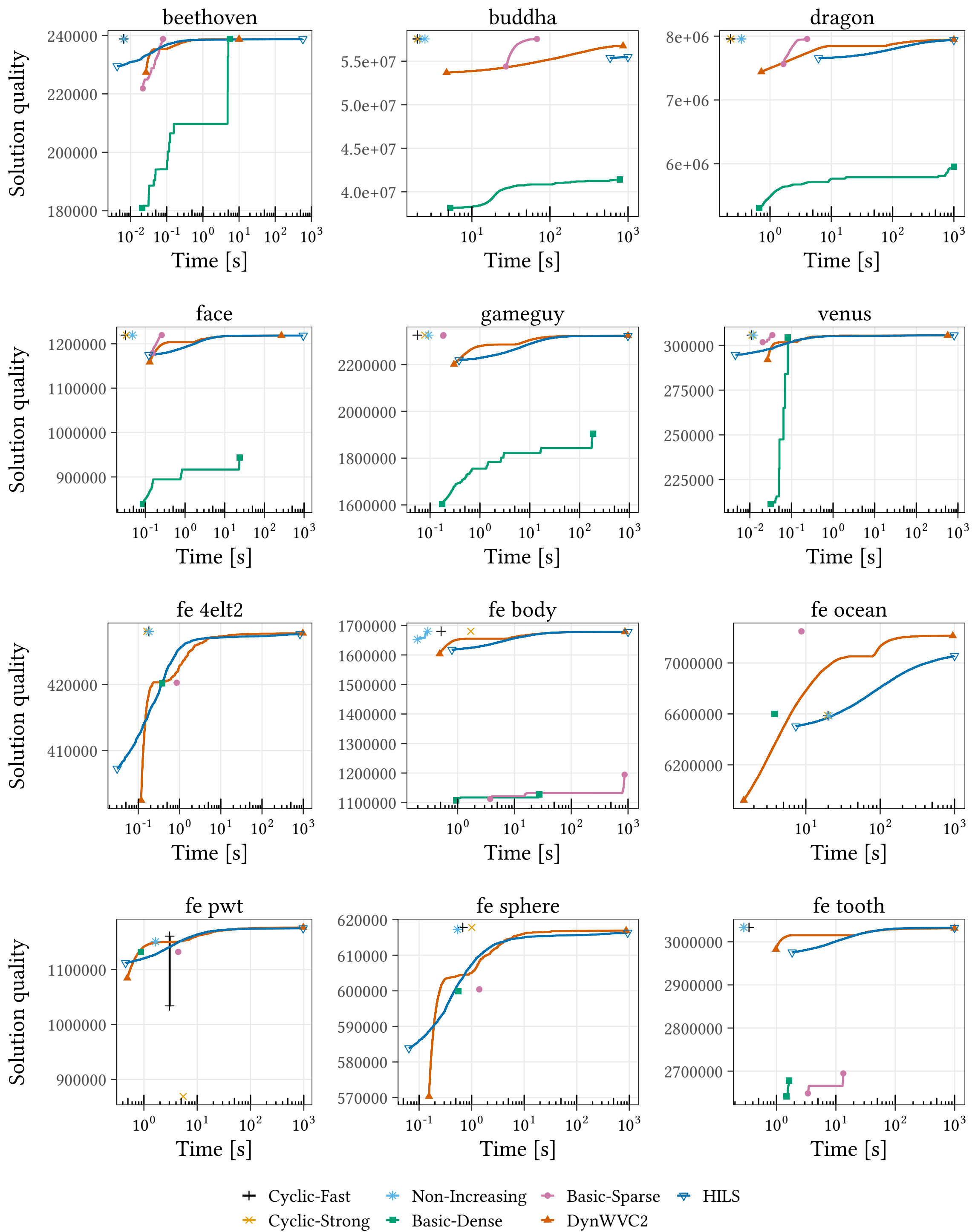



Figure A.9.: Solution quality over time for six mesh instances (upper two rows) and six FE instances (lower two rows).

# Bibliography


[1] F. N. Abu-Khzam, S. Cai, J. Egan, P. Shaw, and K. Wang. “Turbo-charging dominating set with an FPT subroutine: Further improvements and experimental analysis”. In: *International Conference on Theory and Applications of Models of Computation*. Springer. 2017, pp. 59–70. DOI: 10.1007/978-3-319-55911-7_5.

[2] A. A. Ageev. “On Finding Critical Independent and Vertex Sets”. In: *SIAM Journal on Discrete Mathematics* 7.2 (1994), pp. 293–295. DOI: 10.1137/S0895480191217569.

[3] T. Akiba and Y. Iwata. “Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover”. In: *Theoretical Computer Science* 609, Part 1 (2016), pp. 211–225. DOI: 10.1016/j.tcs.2015.09.023.

[4] G. Alexe, P. L. Hammer, V. V. Lozin, and D. de Werra. “Struction revisited”. In: *Discrete applied mathematics* 132.1-3 (2003), pp. 27–46. DOI: 10.1016/S0166-218X(03)00388-3.

[5] D. V. Andrade, M. G. Resende, and R. 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.

[6] F. Ay, M. Kellis, and T. Kahveci. “SubMAP: aligning metabolic pathways with subnetwork mappings”. In: *Journal of computational biology* 18.3 (2011), pp. 219–235. DOI: 10.1089/cmb.2010.0280.

[7] L. Babel. “A fast algorithm for the maximum weight clique problem”. In: *Computing* 52.1 (1994), pp. 31–38. DOI: 10.1007/BF02243394.

[8] E. Balas and C. S. Yu. “Finding a maximum clique in an arbitrary graph”. In: *SIAM Journal on Computing* 15.4 (1986), pp. 1054–1068. DOI: 10.1137/0215075.

[9] L. Barth, B. Niedermann, M. Nöllenburg, and D. Strash. “Temporal Map Labeling: A New Unified Framework with Experiments”. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS ’16. ACM, 2016, 23:1–23:10. DOI: 10.1145/2996913.2996957.

[10] A. E. Brouwer, J. B. Shearer, N. J. A. Sloane, and W. D. Smith. “A new table of constant weight codes”. In: *IEEE Transactions on Information Theory* 36.6 (1990), pp. 1334–1380. DOI: 10.1109/18.59932.

[11] S. Butenko and S. Trukhanov. “Using critical sets to solve the maximum independent set problem”. In: *Operations Research Letters* 35.4 (2007), pp. 519–524. DOI: 10.1016/j.orl.2006.07.004.

[12] S. Cai, W. Hou, J. Lin, and Y. Li. "Improving Local Search for Minimum Weight Vertex Cover by Dynamic Strategies". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018).* 2018, pp. 1412–1418. DOI: 10.24963/ijcai.2018/196.

[13] S. Cai, K. Su, and A. Sattar. "Local search with edge weighting and configuration checking heuristics for minimum vertex cover". In: *Artificial Intelligence* 175.9-10 (2011), pp. 1672–1696. DOI: 10.1016/j.artint.2011.03.003.

[14] L. Chang, W. Li, and W. Zhang. "Computing a near-maximum independent set in linear time by reducing-peeling". In: *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).* 2017, pp. 1181–1196. DOI: 10.1145/3035918.3035939.

[15] J. Chou, J. Kim, and D. Rotem. "Energy-aware scheduling in disk storage systems". In: *31st International Conference on Distributed Computing Systems.* IEEE. 2011, pp. 423–433. DOI: 10.1109/ICDCS.2011.40.

[16] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized algorithms.* Vol. 4. 8. Springer, 2015, pp. 17–20. DOI: 10.1007/978-3-319-21275-3.

[17] J. Dahlum, S. Lamm, P. Sanders, C. Schulz, D. Strash, and R. F. Werneck. "Accelerating Local Search for the Maximum Independent Set Problem". In: *Experimental Algorithms (SEA 2016).* Ed. by A. V. Goldberg and A. S. Kulikov. Vol. 9685. LNCS. Springer, 2016, pp. 118–133. DOI: 10.1007/978-3-319-38851-9_9.

[18] E. D. Dolan and J. J. Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical programming* 91.2 (2002), pp. 201–213. DOI: 10.1007/s101070100263.

[19] C. Ebenegger, P. Hammer, and D. De Werra. "Pseudo-Boolean functions and stability of graphs". In: *North-Holland mathematics studies.* Vol. 95. Elsevier, 1984, pp. 83–97. DOI: 10.1016/S0304-0208(08)72955-4.

[20] D. Ferizovic, D. Hespe, S. Lamm, M. Mnich, C. Schulz, and D. Strash. "Engineering Kernelization for Maximum Cut". In: *Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX).* SIAM. 2020, pp. 27–41. DOI: 10.1137/1.9781611976007.3.

[21] M. R. Garey, D. S. Johnson, and L. Stockmeyer. "Some Simplified NP-Complete Problems". In: *Proceedings of the 6th ACM Symposium on Theory of Computing (STOC '74).* ACM, 1974, pp. 47–63. DOI: 10.1145/800119.803884.

[22] A. Gemsa, M. Nöllenburg, and I. Rutter. "Evaluation of Labeling Strategies for Rotating Maps". English. In: *Experimental Algorithms (SEA'14).* Vol. 8504. LNCS. Springer, 2014, pp. 235–246. DOI: 10.1007/978-3-319-07959-2_20.

[23] J. R. Gilbert, G. L. Miller, and S.-H. Teng. "Geometric mesh partitioning: Implementation and experiments". In: *SIAM Journal on Scientific Computing* 19.6 (1998), pp. 2091–2110. DOI: 10.1109/IPPS.1995.395965.

[24] P. L. Hammer, N. V. R. Mahadev, and D. de Werra. “The struction of a graph: Application to CN-free graphs”. In: *Combinatorica* 5.2 (1985), pp. 141–147. DOI: 10.1007/BF02579377.

[25] P. L. Hammer, N. V. Mahadev, and D. de Werra. “Stability in CAN-free graphs”. In: *Journal of Combinatorial Theory, Series B* 38.1 (1985), pp. 23–30. DOI: 10.1016/0095-8956(85)90089-9.

[26] M. Henzinger, A. Noe, and C. Schulz. “Faster Parallel Multiterminal Cuts”. In: *arXiv preprint arXiv:2004.11666* (2020).

[27] M. Henzinger, A. Noe, and C. Schulz. “Shared-memory exact minimum cuts”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2019, pp. 13–22. DOI: 10.1109/IPDPS.2019.00013.

[28] M. Henzinger, A. Noe, C. Schulz, and D. Strash. “Finding All Global Minimum Cuts In Practice”. In: *arXiv preprint arXiv:2002.06948* (2020).

[29] M. Henzinger, A. Noe, C. Schulz, and D. Strash. “Practical minimum cut algorithms”. In: *Journal of Experimental Algorithmics (JEA)* 23 (2018), pp. 1–22. DOI: 10.1145/3274662.

[30] D. Hespe, S. Lamm, C. Schulz, and D. Strash. “WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track”. In: *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*. SIAM. 2020, pp. 1–11. DOI: 10.1137/1.9781611976229.1.

[31] D. Hespe, C. Schulz, and D. Strash. “Scalable Kernelization for Maximum Independent Sets”. In: *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2018, pp. 223–237. DOI: 10.1137/1.9781611975055.19.

[32] K. W. Hoke and M. Troyon. “The struction algorithm for the maximum stable set problem revisited”. In: *Discrete Mathematics* 131.1-3 (1994), pp. 105–113. DOI: 10.1016/0012-365X(94)90377-8.

[33] S. Lamm, C. Schulz, D. Strash, R. Williger, and H. Zhang. “Exactly solving the maximum weight independent set problem on large real-world graphs”. In: *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2019, pp. 144–158. DOI: 10.1137/1.9781611975499.12.

[34] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. http://snap.stanford.edu/data. 2014.

[35] C.-M. Li, H. Jiang, and F. Manyà. “On minimization of the number of branches in branch-and-bound algorithms for the maximum clique problem”. In: *Computers & Operations Research* 84 (2017), pp. 1–15. DOI: 10.1016/j.cor.2017.02.017.

[36] R. Li, S. Hu, S. Cai, J. Gao, Y. Wang, and M. Yin. “NuMWVC: A novel local search for minimum weighted vertex cover problem”. In: *Journal of the Operational Research Society* (2019), pp. 1–12. DOI: 10.1080/01605682.2019.1621218.

[37] Y. Li, S. Cai, and W. Hou. “An Efficient Local Search Algorithm for Minimum Weighted Vertex Cover on Massive Graphs”. In: *Asia-Pacific Conference on Simulated Evolution and Learning (SEAL 2017)*. Vol. 10593. LNCS. 2017, pp. 145–157. DOI: 10.1007/978-3-319-68759-9_13.

[38] L. Lovász and M. D. Plummer. *Matching theory*. Vol. 29. 1986, pp. 471–482.

[39] V. V. Lozin. “Conic reduction of graphs for the stable set problem”. In: *Discrete Mathematics* 222.1-3 (2000), pp. 199–211. DOI: 10.1016/S0012-365X(99)00408-2.

[40] T. Ma and L. J. Latecki. “Maximum weight cliques with mutex constraints for video object segmentation”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2012, pp. 670–677. DOI: 10.1109/CVPR.2012.6247735.

[41] F. Mascia, E. Cilia, M. Brunato, and A. Passerini. “Predicting structural and functional sites in proteins by searching for maximum-weight cliques”. In: *Twenty-Fourth AAAI Conference on Artificial Intelligence*. 2010.

[42] K. Mehlhorn and P. Sanders. *Algorithms and data structures: The basic toolbox*. Springer Science & Business Media, 2008, pp. 133–139.

[43] B. Nogueira, R. G. S. Pinheiro, and A. Subramanian. “A hybrid iterated local search heuristic for the maximum weight independent set problem”. In: *Optimization Letters* 12.3 (2018), pp. 567–583. DOI: 10.1007/s11590-017-1128-7.

[44] K. J. Nurmela, M. K. Kaikkonen, and P. Ostergard. “New constant weight codes from linear permutation groups”. In: *IEEE Transactions on Information Theory* 43.5 (1997), pp. 1623–1630. DOI: 10.1109/18.623163.

[45] *OpenStreetMap*. URL: https://www.openstreetmap.org.

[46] P. R. Östergård. “A fast algorithm for the maximum clique problem”. In: *Discrete Applied Mathematics* 120.1-3 (2002), pp. 197–207. DOI: 10.1016/S0166-218X(01)00290-6.

[47] A. S. P. Pedersen, P. D. Vestergaard, et al. “Bounds on the number of vertex independent sets in a graph”. In: *Taiwanese Journal of Mathematics* 10.6 (2006), pp. 1575–1587. DOI: 10.11650/twjm/1500404576.

[48] H. Prodinger and R. Tichy. “Fibonacci numbers of graphs”. In: *The Fibonacci Quarterly* 20.1 (1982), pp. 16–21.

[49] S. Rebennack, M. Oswald, D. O. Theis, H. Seitz, G. Reinelt, and P. M. Pardalos. “A branch and cut solver for the maximum stable set problem”. In: *Journal of combinatorial optimization* 21.4 (2011), pp. 434–457. DOI: 10.1007/s10878-009-9264-3.

[50] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. “Efficient traversal of mesh edges using adjacency primitives”. In: *ACM Transactions on Graphics (TOG)* 27.5 (2008), pp. 1–9. DOI: 10.1145/1409060.1409097.

[51] P. Sanders and C. Schulz. “Distributed evolutionary graph partitioning”. In: *Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2012, pp. 16–29. DOI: 10.1137/1.9781611972924.2.

[52] R. E. Tarjan and A. E. Trojanowski. “Finding a maximum independent set”. In: *SIAM Journal on Computing* 6.3 (1977), pp. 537–546. DOI: 10.1137/0206038.

[53] J. Trimble. Version v0.2. 2017. DOI: 10.5281/zenodo.848647. URL: https://doi.org/10.5281/zenodo.848647.

[54] J. S. Warren and I. V. Hicks. “Combinatorial branch-and-bound for the maximum weight independent set problem”. 2006. URL: https://www.caam.rice.edu/~ivhicks/jeff.rev.pdf.

[55] D. Warrier. “A branch, price, and cut approach to solving the maximum weighted independent set problem”. PhD thesis. Texas A&M University, 2007. DOI: 1969.1/5814.

[56] D. Warrier, W. E. Wilhelm, J. S. Warren, and I. V. Hicks. “A branch-and-price approach for the maximum weight independent set problem”. In: *Networks: An International Journal* 46.4 (2005), pp. 198–209. DOI: 10.1002/net.20088.

[57] Q. Wu and J.-K. Hao. “Solving the winner determination problem via a weighted maximum clique heuristic”. In: *Expert Systems with Applications* 42.1 (2015), pp. 355–365. DOI: 10.1016/j.eswa.2014.07.027.

[58] H. Xu, T. S. Kumar, and S. Koenig. “A new solver for the minimum weighted vertex cover problem”. In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer. 2016, pp. 392–405. DOI: 10.1007/978-3-319-33954-2_28.

[59] X. Xu, S. Tang, and P.-J. Wan. “Maximum weighted independent set of links under physical interference model”. In: *International Conference on Wireless Algorithms, Systems, and Applications*. Springer. 2010, pp. 68–74. DOI: 10.1007/978-3-642-14654-1_8.

[60] R. Zamprogno and A. R. Amaral. “An efficient approach for large scale graph partitioning”. In: *Journal of combinatorial optimization* 13.4 (2007), p. 289. DOI: 10.1007/s10878-006-9026-4.

[61] B. Zavalnij. “Generalizing struction kernelization method for maximum weighted independent set”. private communication. 2019.