There and Back Again: On Applying Data Reduction Rules by Undoing Others

Aleksander Figiel
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

Vincent Froese
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

André Nichterlein
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

Rolf Niedermeier
Algorithmics and Computational Complexity, Technische Universität Berlin, Germany

Abstract

Data reduction rules are an established method in the algorithmic toolbox for tackling computationally challenging problems. A data reduction rule is a polynomial-time algorithm that, given a problem instance as input, outputs an equivalent, typically smaller instance of the same problem. The application of data reduction rules during the preprocessing of problem instances allows in many cases to considerably shrink their size, or even solve them directly. Commonly, these data reduction rules are applied exhaustively and in some fixed order to obtain irreducible instances. It was often observed that by changing the order of the rules, different irreducible instances can be obtained. We propose to "undo" data reduction rules on irreducible instances, by which they become larger, and then subsequently apply data reduction rules again to shrink them. We show that this somewhat counter-intuitive approach can lead to significantly smaller irreducible instances. The process of undoing data reduction rules is not limited to "rolling back" data reduction rules applied to the instance during preprocessing. Instead, we formulate so-called backward rules, which essentially undo a data reduction rule, but without using any information about which data reduction rules were applied to it previously. In particular, based on the example of Vertex Cover we propose two methods applying backward rules to shrink the instances further. In our experiments we show that this way smaller irreducible instances consisting of real-world graphs from the SNAP and DIMACS datasets can be computed.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Parameterized complexity and exact algorithms; Theory of computation → Branch-and-bound

Keywords and phrases Kernelization, Preprocessing, Vertex Cover

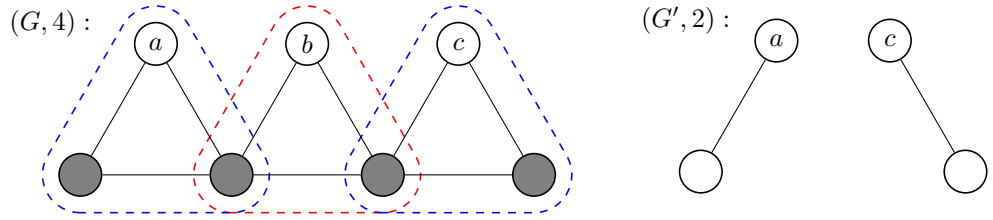
Digital Object Identifier 10.4230/LIPIcs.ESA.2022.53

Related Version *A full version of the paper is available at*: `https://arxiv.org/abs/2206.14698` [10]

Supplementary Material *Software (Implementation)*: `https://git.tu-berlin.de/afigiel/undo-vc-drr`, archived at `swh:1:dir:54f57455bc1dc965f9315a1cc800cfd768bbdac3`

Funding *Aleksander Figiel*: Supported by DFG project "MaMu" (NI369/19).

Acknowledgements This work is based on the first author's master's thesis.
In memory of Rolf Niedermeier, our colleague, friend, and mentor, who sadly passed away before this paper was published.



(a) An instance $(G, 4)$ for Vertex Cover. A vertex cover of size four is indicated by the gray vertices.

(b) The outcome $(G', 2)$ of applying the Triangle Rule to b.

Figure 1 An example of how the order of reduction rules can affect the final instance. A Vertex Cover instance with $k = 4$ is depicted left. By applying the Triangle Rule to b (red triangle), we obtain a graph with two edges (right). However, applying the rule to a and then c (therefore removing the two blue triangles) yields the smaller graph consisting only of the vertex b.

1 Introduction

Kernelization by means of applying data reduction rules is a powerful (and often essential) tool for tackling computationally difficult (e. g. NP-hard) problems in theory and in practice [11, 2]. A data reduction rule is a polynomial-time algorithm that, given a problem instance as input, outputs an "equivalent" and often smaller instance of the same problem. One may think of data reduction as identifying and removing "easy" parts of the problem, leaving behind a smaller instance containing only the more difficult parts. This instance can be significantly smaller than the original instance [1, 3, 16, 13, 18] which makes other methods like branch&bound algorithms a viable option for solving it. In this work, we apply existing data reduction rules "backwards", that is, instead of smaller instances we produce (slightly) larger instances. The hope herein is, that this alteration of the instance allows subsequently applied data reduction rules to further shrink the instance, thus, producing even smaller instances than with "standard" application of data reduction rules.

We consider the NP-hard Vertex Cover – the primary "lab animal" in parameterized complexity theory [9] – to illustrate our approach and to exemplify its strengths.

Vertex Cover [12]

Input: An undirected graph $G = (V, E)$ and $k \in \mathbb{N}$.
Question: Is there a set $S \subseteq V$, $|S| \leq k$, covering all edges, i. e., $\forall e \in E \colon e \cap S \neq \emptyset$?

Vertex Cover is a classic problem of computational complexity theory and one of Karp's 21 NP-complete problems [15]. We remark that for presentation purposes we use the decision version of Vertex Cover. All our results transfer to the optimization version (which our implementation is build for).

To explain our approach, assume that all we have is the following data reduction rule:

▶ **Reduction Rule 1** (Triangle Rule [9]). *Let $(G = (V, E), k)$ be an instance of* Vertex Cover *and $v \in V$ a vertex with exactly two neighbors u and w. If the edge $\{u, w\}$ exists, then delete v, u, and w from the graph (and their incident edges), and decrease k by two.*

As illustrated in Figure 1, there are two options to apply Rule 1 for the instance $(G, 4)$. Picking the "bad" option, that is, applying it to b yields the instance $(G', 2)$. Note that (the correctness of) Rule 1 implies that $(G, 4)$ and $(G', 2)$ are "equivalent", that is, either both of them are yes-instances or none of them are. Hence, if we have the instance $(G', 2)$ on the right side (either through the "bad" application of Rule 1 or directly as input), then we can

apply Rule 1 *"backwards"* and obtain the equivalent but larger instance $(G, 4)$ on the left side. Then, by applying Rule 1 to a and c, we can arrive at the edge-less graph $(\{b\}, \emptyset)$, thus "solving" the triangle-free instance $(G', 2)$ by only using the Triangle Rule.

More formally, the setting can be described as follows: A data reduction rule for a problem L is a polynomial-time algorithm which *reduces* an instance x to an equivalent instance x', that is, $x \in L$ if and only if $x' \in L$. A set of data reduction rules thus implicitly partitions the space of all instances into classes of equivalent instances (two instances are in the same equivalence class if one of them can be obtained from the other by applying a subset of the data reduction rules). The more data reduction rules we have, the fewer and larger equivalence classes we have. Now, the overall goal of data reduction is to find the *smallest* instance in the same equivalence class. We demonstrate two approaches tailored towards (but not limited to) graph problems to tackle this task.

Let us remark that while there are some analogies to the branch&bound paradigm (searching for a solution in a huge search space), there are also notable differences: A branching rule creates several instances of which at least one is guaranteed to be equivalent to the original one. The problem is that, a priori, it is not known which of these instances is the equivalent one. Hence, one has to "solve" all instances before learning the solution. In contrast, our setting allows stopping at *any* time as the currently handled instance is *guaranteed* to be equivalent to the starting instance. This allows for considerable flexibility with respect to possible combinations with other approaches like heuristics, approximation or exact algorithms.

Related Work. Fellows et al. [9] are closest to our work. They propose a method for automated discovery of data reduction rules looking at rules that replace a small subgraph by another one. They noticed that if a so called *profile* (which is a vector of integers) of the replaced subgraph and of the one taking its place only differ by a constant in each entry, then this replacement is a data reduction rule. To then find data reduction rules, one can enumerate all graphs up to a certain size and compute their profile vectors. The downside of this approach is that in order to apply the automatically found rules one has solve a (computationally challenging) subgraph isomorphism problem or manually design new algorithms for each new rule.

Vertex Cover is extensively studied from the the viewpoint of data reduction and kernelization; see Fellows et al. [9] for an overview. Akiba and Iwata [3] and Hespe et al. [14] provide exact solvers that include an extensive list of data reduction rules. The solver of Hespe et al. [14] won the exact track for Vertex Cover at the 4th PACE implementation challenge [7]. We provide a list of data reduction rules for Vertex Cover in the full version [10].

Alexe et al. [4] experimentally investigated by how much the so-called Struction data reduction rule for Independent Set can shrink small random graphs. The Struction data reduction rule can always be applied to any graph and decreases the stability number[1] of a graph by one, but may increase the number of vertices quadratically each time it is applied. Gellner et al. [13] proposed a modification of the Struction rule for the Maximum Weighted Independent Set problem. They first restricted themselves to only applying data reduction rules if they do not increase the number vertices in the graph, which they call the reduction phase. They then compared this method to an approach which allows

[1] An independent set is a set of pairwise nonadjacent vertices. The stability number or the independence number of a graph G is the size of a maximum independent set of G.

their modified Struction rule to also increase the number of vertices in the graph by a small fraction, which they call the blow-up phase. The experiments showed, that repetitions of the reduction and blow-up phase can significantly shrink the number of vertices compared to just the reduction phase.

Ehrig et al. [8] defined the notion of confluence from rewriting systems theory for kernelization algorithms. Intuitively, confluence in kernelization means that the result of applying a set of data reduction rules exhaustively to the input always results in the same instance, up to isomorphism, regardless of the order in which the rules were applied. It turns out that for our approach to work we require non-confluent data reduction rules.

Our Results. In Section 3, we provide two concrete methods to apply existing data reductions rules "backwards" and "forwards" in order to shrink the input as much as possible. We implemented these methods and applied them on a wide range of data reduction rules for Vertex Cover. Our experimental evaluations are provided in Section 4 where we use our implementation on instances where the known data reductions rules are not applicable. Our implementation can also be used to preprocess a given graph G and it returns the smallest found kernel K after a user specified amount of time. Moreover, the implementation can translate a provided solution S_K for the kernel into a solution S_G for the initial instance such that $|S_G| \leq |S_K| + d$ where $d := \tau(G) - \tau(K)$ is the difference between the sizes of minimum vertex covers of G and K. Thus, if a minimum vertex cover for K is provided it will be translated into a minimum vertex cover for G.

2 Preliminaries

We use standard notation from graph theory and data reduction. In this work, we only consider simple undirected graphs G with vertex set $V(G)$ and edge set $E(G) \subseteq \{\{v, w\} \mid v, w \in V(G), v \neq w\}$. We denote by n and m the number of vertices and edges, respectively. For a vertex $v \in V(G)$ the open (closed) neighborhood is denoted with $N_G[v]$ ($N_G(v)$). For a vertex subset $S \subseteq V(G)$ we set $N_G[S] := \bigcup_{v \in S} N_G[v]$. When in context it is clear which graph is being referred to, the subscript G will be omitted in the subscripts.

Data Reduction Rules. We use notions from kernelization in parameterized algorithmics [11]. However, we simplify the notation to unparameterized problems. A data reduction rule for a problem $L \subseteq \Sigma^*$ is a polynomial-time algorithm, which *reduces* an instance x to an equivalent instance x'. We call an instance x *irreducible* with respect to a data reduction rule, if the data reduction rule does not change the instance x any further (that is, $x' = x$). The property that the data reduction rule returns an equivalent instance is called *safeness*. We call an instance obtained from applying data reduction rules *kernel*.

Often, data reduction rules can be considered nondeterministic, because a data reduction rule could change the input instance in a variety of ways (e. g., see the example in Section 1). To highlight this effect and to avoid confusion, we introduce the term *forward rule*. A forward rule is a subset $R_{\mathcal{A}} \subseteq \Sigma^* \times \Sigma^*$ associated with a nondeterministic polynomial-time algorithm $\mathcal{A}$, where $(x, y) \in R_{\mathcal{A}}$ if and only if y is one of the possible outputs of $\mathcal{A}$ on input x. Intuitively, a forward rule captures all possible instances that can be derived from the input instance by applying a data reduction rule a single time. To define what it means to "undo" a reduction rule, we introduce the term *backward rule*. A backward rule is simply the converse relation $R_{\mathcal{A}}^{-1} := \{(y, x) \mid (x, y) \in R_{\mathcal{A}}\}$ of some forward rule $R_{\mathcal{A}}$.

Confluence. A set of data reduction rules is said to be terminating, if for all instances $\mathcal{I}$, the data reduction rules in the set cannot be applied to the instance $\mathcal{I}$ infinitely many times. A set of terminating data reduction rules is said to be confluent if any exhaustive way of applying the rules yields a unique irreducible instance, up to isomorphism [8]. It is not hard to see that given a set of confluent data reduction rules, undoing any of them is of no use, because subsequently applying the data reduction rules will always result in the same instance.

▶ **Lemma 2.1.** *Let $\mathcal{R}$ be a confluent set of forward rules, $\mathcal{I}$ a problem instance, and $\mathcal{I}^{\mathcal{R}}$ the unique instance obtained by applying the rules in $\mathcal{R}$ to $\mathcal{I}$.*

Further let $\overline{\mathcal{R}} = \{R^{-1} \mid R \in \mathcal{R}\}$ be the set of backward rules corresponding to $\mathcal{R}$. Let $\mathcal{I}^{-}$ be an instance that was derived by applying some rules from $\mathcal{R} \cup \overline{\mathcal{R}}$. Then applying the rules in $\mathcal{R}$ exhaustively in any order to $\mathcal{I}^{-}$ will yield an instance isomorphic to $\mathcal{I}^{\mathcal{R}}$.

Proof. We prove the lemma by induction over the number of times backward rules were applied to obtain $\mathcal{I}^{-}$.

Base case: if no backward rules were applied to obtain $\mathcal{I}^{-}$, then by exhaustively applying $\mathcal{R}$ we will obtain an instance isomorphic to $\mathcal{I}^{\mathcal{R}}$.

Inductive step: Assume only n backward rules were applied to obtain $\mathcal{I}^{-}$. Consider the sequence $\mathcal{I} = \mathcal{I}_1, \mathcal{I}_2 \ldots, \mathcal{I}_{\ell-1}, \mathcal{I}_\ell = \mathcal{I}^{-}$ of instances which were on the way from $\mathcal{I}$ to $\mathcal{I}^{-}$ during the application of the rules in $\mathcal{R} \cup \overline{\mathcal{R}}$. Further let $\mathcal{I}_i$ be the instance in the sequence after applying the n'th backward rule R^{-1}. After applying R to $\mathcal{I}_i$ the instance $\mathcal{I}_{i-1}$ can be obtained, which was obtained by using only $n-1$ backward rules. Because after R^{-1} only rules in $\mathcal{R}$ were applied, which are confluent, we may instead assume that the first rule which was applied after R^{-1} was R. Consequently, by the induction hypothesis an instance isomorphic to $\mathcal{I}^{\mathcal{R}}$ will be obtained by exhaustively applying the rules in $\mathcal{R}$. ◀

3 Two Methods for Achieving Smaller Kernels

We apply data reduction rules "back and forth" to obtain an equivalent instance as small as possible. This gives rise to a huge search space for which exhaustive search is prohibitively expensive. Thus, some more sophisticated search procedures are needed. In this section, we propose two approaches which we call the Find and the Inflate-Deflate method. We implemented and tested both approaches; the experimental results are presented in Section 4.

The Find method (Section 3.1) shrinks the naive search tree with heuristic pruning rules in order to identify sequences of forward and backward rules, which when applied to the input instance, produce a smaller equivalent instance. We employ this method primarily to find such sequences which are *short*, so that those sequences may actually be used to formulate *new* data reduction rules. It naturally has a *local* flavor in the sense that changes of one iteration are bound to a (small) part of the input graph.

The Inflate-Deflate method (Section 3.2) is much less structured. It randomly applies backward rules until the instance size increased by a fixed percentage. Afterwards, all forward rules are applied exhaustively. If the resulting instance is smaller, then the process is repeated; otherwise, all changes are reverted.

3.1 Find Method

For finding sequences of forward and backward rules which when applied to the input instance produce a smaller instance, we propose a structured search approach based on recursion. Let $\mathcal{I}$ be the input instance and let $\mathcal{F}_{\mathcal{I}}$ be the set of all instances reachable via one forward

or backward rule, i. e., $\mathcal{F}_{\mathcal{I}} = \{\mathcal{I}' \mid (\mathcal{I}, \mathcal{I}') \in R \text{ for any forward or backward rule } R\}$. If the input instance is irreducible with respect to the set of forward rules, then only backward rules will be applicable. We branch into $|\mathcal{F}_{\mathcal{I}}|$ cases where, for each $\mathcal{I}' \in \mathcal{F}_{\mathcal{I}}$, we try to recursively find forward and backward rules applicable to $\mathcal{I}'$ and branch on each of them. This is repeated until some maximum recursion depth is reached or the sequence of forward and backward rule applications results in a smaller instance. In the latter case, the sequence of applied rules can be thought of as a new data reduction rule.

Note that the search space is immense: For example, consider the backward rule corresponding to Rule 1, which inserts three vertices u, v, and w, makes them pairwise adjacent, and inserts an arbitrary set of edges between u and v and the original vertices. Thus, for each original vertex, there are four options (make it adjacent to u, to v, to u and v, or neither). This results in 4^n options for applying just this one single backward rule. Hence, it is clear that we have to introduce suitable methods to cut off large parts of the resulting search tree. To this end, we heavily rely on the observation that many reduction rules have a "local flavor".

Region of Interest. To avoid a very large search space we only consider applying forward and backward rules "locally". For this, we define a "region of interest", which for graph problems is a set $X \subseteq V$. Any forward or backward rule must only be applied within the region of interest. We initially start with a very modest region of interest, namely $X = \{v\}$ for all $v \in V$. Thus, we work with n regions of interest per graph, each one considered separately.

Forward and backward rules are allowed to leave the region of interest only if at least one vertex that is "relevant" for the rule is within the region of interest. Moreover, the region of interest is allowed to "grow" as rules are applied. This is because applications of rules might cause further rules to become applicable. For example, rules might become applicable to the neighbors of vertices modified by the previously applied rules.

Specifically, let M be the set of "modified" vertices, which are new vertices or vertices which gained or lost an edge as a result of applying a forward or backward rule, and let D be the set of vertices it deleted. In the case of Vertex Cover, we suggest to expand the region of interest to $(X \cup N[M]) \setminus D$ after each rule application. The majority of forward rules for Vertex Cover are "neighborhood based". Of course, the region of interest could be expanded even further, e. g., by extending it by $N^2[M]$ instead, but this will of course increase the search space. With a larger region of interest we might find more reduction rules, but at the cost of higher running time. Additionally, the found reductions may be more complex, and modify a large subgraph and are therefore difficult to analyze or implement.

We will call the above method, which recursively applies forward and backward rules one by one restricted to only the region of interest, the Find method. Note that we "accept" a sequence of forward and backward rules if it decreases the number of vertices or the parameter k, without increasing either. We have done so to find only "nice" data reduction rules where there is no trade-off between decreasing the parameter k or the number of vertices. However, different conditions to "accept" a sequence of forward and backward rules are also possible. For example, by only requiring that the number of vertices or edges is decreased.

Graph modification. Our implementation of the Find method outputs sequences of rules which are able to shrink the graph. However, just knowing which rules and in what order they were applied may not be very helpful in understanding the changes made by the rules. Specifically, it does not show how the rules were applied.

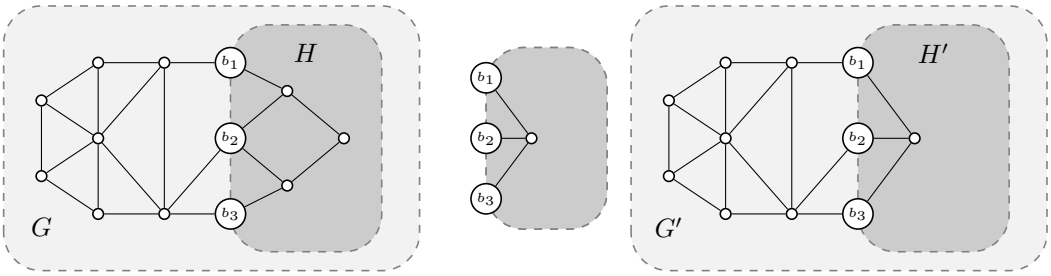


Figure 2 A graph modification example corresponding to the application of the Degree-2 Folding Rule that applies to vertices with exactly two non-adjacent neighbors (the rule merges the degree-two vertex and its neighbors). The set $B = \{b_1, b_2, b_3\}$ is the boundary of the graph modification. Notice that only vertices in B (the boundary) are adjacent to both vertices in $V(H)$ and $V(G) \setminus V(H)$.

For this reason, we introduce the notion of a *graph modification* based on the ideas by Fellows et al. [9]. A graph modification encodes how a single or multiple data reduction rules have changed a graph. We say the *boundary* of a subgraph H of a graph G is the set B of all vertices in $V(H)$ whose neighborhood in G contains vertices in $V(G) \setminus V(H)$.

▶ **Definition 3.1.** *A graph modification is a 4-tuple of graphs (G, H, H', G') with the following properties*

- $B = V(H') \cap V(G)$,
- *H is a subgraph of G with boundary B, and*
- *G' is derived from G by deleting all vertices in $V(H) \setminus B$ and all edges among B from G and then adding the vertices in $V(H') \setminus B$ and adding all edges from H'.*

See Figure 2 for an example of a graph modification. The Find method can be extended such that in addition to printing rule sequences it also outputs the graph modifications corresponding to each application of a rule from the found sequences. This can be achieved by keeping track of newly created or deleted vertices and edges by the applied rules.

Isomorphism. It may happen that two or more rule sequences change an instance in the same way, that is, they produce isomorphic graphs from the same input instance. In order to avoid double counting, we implemented an isomorphism test to avoid these issues; see the full version [10] for details.

The Find and Reduce Method. The Find and Reduce Method is just a small variation of Find. Instead of only searching for sequences of forward and backward rules which shrink the instance, upon finding such a sequence it is also immediately applied to the instance. The search for more sequences continues with the smaller instance. This method serves the dual purpose of both finding sequences of rules which shrink the instance (and therefore also finding reduction rules), but also that of producing a smaller irreducible instance. Another potential advantage of this method is that it finds only the rules which were used to produce the smaller instance, and may therefore be more practical than the ones found by Find. Recall, that Find only searches for reduction rules applicable directly to the input instance. Perhaps by applying a single such sequence, different sequences are needed to shrink the remaining instance further.

3.2 Inflate-Deflate

Both the Find and the Find and Reduce method only change a small part of the instance. It may, however, be necessary to change large parts of an instance before it can be shrunk to a size smaller than it was originally. For this reason, we propose the Inflate-Deflate inspired by the Cyclic Blow-Up Algorithm by Gellner et al. [13].

Essentially, Inflate-Deflate iteratively runs two phases: First, in the inflation phase, randomly applies a set of backward rules to the instance until it becomes some fixed percentage α larger than it was initially. Then, in the deflation phase, exhaustively apply a set of forward rules and repeat with the inflation phase again. Our implementation has the termination condition $|V| = 0$ which may never be met. Thus a timeout or limit on the number of iterations has to be specified. The inflation factor α can be freely set to any value greater zero. We investigate the effect of different inflation factors in Section 4 for values of α between 10% and 50%.

In our deflate procedure, we apply the set of forward rules exhaustively in a particular way: An applicable forward rule is randomly chosen, and then it is randomly applied to the instance, but only once. Afterwards another applicable rule is chosen randomly, and this is repeated until the instance becomes irreducible with respect to all forward rules. This randomized exhaustive application of the forward rules ensures that the rules are not applied in a predefined way, and different "interactions" of the rules are tested. For example, consider the case where in the inflate phase the Backward Degree-2 Folding Rule was applied, then likely one would not want to immediately exhaustively apply the (Forward) Degree-2 Folding Rule (see Figure 2) which could in effect directly cancel the changes made by that backward rule. Furthermore, in this way, each of the forward rules has a chance to be applied. This avoids any potential problems due to an inconvenient fixed rule order.

Because large sections of an instance are modified at once, no short sequences of forward and backward rules can be extracted from this method. As a result, it is unlikely that new reduction rules could be learned this way. However, the method produces smaller irreducible instances as we will see in Section 4.

Local Inflate-Deflate. Within Inflate-Deflate it may happen that if we inflate the instance and then deflate it again, often the resulting instance is larger. In such cases the number of "negative" changes to the instance outweigh the number of "positive" changes. To increase the success probability one may try to lower the inflation factor, however then it can also happen that positive changes are less likely.

An alternate way to try to increase the success probability, is to apply backward rules within a randomly chosen subgraph rather than the whole graph. For example, this subgraph could be the set of all vertices with some maximum distance to a randomly chosen vertex. Backward rules are then applied until the subgraph becomes larger by a factor of α, instead of the whole graph.

4 Experimental Evaluation

In this section, we describe the experiments which we performed based on an implementation of the methods described in Section 3.

Table 1 A glossary of the forward and backward rules which were used in our implementation. Note that we apply the Struction Rule only if it does not increase k. In the columns named alias we provide shortened names for the rules. We refer to the full version of the paper [10] for detailed descriptions of these rules.

Forward rules		Backward rules	
Alias	Full name	Alias	Full name
Deg0	Degree-0	Undeg2	Backward Degree-2 Folding
Deg1	Degree-1	Undeg3	Backward Degree-3 Independent Set
Deg2	Degree-2 Folding	Uncn	Backward 2-Clique Neighborhood (special case)
Deg3	Degree-3 Independent Set	Undom	Backward Domination
Dom	Domination	Ununconf	Backward Unconfined
Unconf	Unconfined-κ ($\kappa = 4$)	OE_Ins	Optional Edge Insertion
Desk	Desk		
CN	2-Clique Neighborhood		
OE_Del	Optional Edge Deletion		
Struct	Struction ($k' \leq k$)		
Magnet	Magnet		
LP	LP		

4.1 Setup

Computing Environment. All our experiments were run on a machine running Ubuntu 18.04 LTS with the Linux 4.15 kernel. The machine is equipped with an Intel® Xeon® W-2125 CPU, with 4 cores and 8 threads[2] clocked at 4.0 GHz and 256GB of RAM.

Datasets. For our experiments we used three different datasets: DIMACS, SNAP and PACE; the lists of graphs are given in the full version [10]. The DIMACS and SNAP datasets are commonly used for graph-based problems, including Vertex Cover [3, 16, 13]. We have used the instances from the 10th DIMACS Challenge [5], specifically from the Clustering, Kronecker, Co-author and Citation, Street Networks, and Walshaw subdatasets. In total these are 82 DIMACS instances. From the SNAP Dataset Collection [17] we have used the graphs from from the Social, Ground-Truth Communities, Communication, Collaboration, Web, Product Co-purchasing, Peer-to-peer, Road, Autonomous systems, Signed and Location subdatasets. In total we obtained 52 SNAP instances. Additionally, we used a dataset which was used specifically for benchmarking Vertex Cover solvers in the 2019 PACE Challenge [7]. We used the set of 100 private instances [6], which were used for scoring submitted solvers.

Preprocessing and Filtering. We apply some preprocessing to our datasets. We obtain simple, undirected graphs by ignoring any potential edge direction or weight information from the instances and by deleting self-loops. To these graphs we apply the forward rules, see Table 1 and the full version [10] for an overview: Deg1, Deg2, Deg3, Unconf, Cn, LP, Struct, Magnet and Oe_delete exhaustively in the given order. We note that the kernels obtained this way always had fewer vertices than the kernels obtained with the data reduction suite used by Akiba and Iwata [3] and also Hespe et al. [14].

[2] All our implementations are single-threaded.

We filter out graphs that became empty as a result of applying these rules. These were 41 DIMACS, 33 SNAP and 12 PACE instances. Furthermore, we discard graphs which after applying these rules still had more than 50,000 vertices. These were 12 DIMACS and 3 SNAP instances. Because the PACE instances may also contain instances from the other two datasets, we have tested the graphs for isomorphism. We have found one PACE instance to be isomorphic to a SNAP instance (p2p-Gnutella09), which was already excluded, because it was shrunk to an empty graph.

In total, we are left with 31 DIMACS, 16 SNAP and 88 PACE graphs with at most 50,000 vertices – all of these graphs are irreducible with respect to the forward rules. When referring to the datasets DIMACS, SNAP and PACE we will be referring to these kernelized and filtered instances.

Implementation. The major parts of our implementation are written in C++11 and compiled using version 7.5 of g++ using the -O2 optimization flag. Smaller parts, such as scripts for visualization or automation were written in Python 3.6 or Bash. We provide the source code for our implementation at `https://git.tu-berlin.de/afigiel/undo-vc-drr`. This implementation contains our Find and Inflate-Deflate method, together with the two small variations Find and Reduce, and local Inflate-Deflate. Find uses the local isomorphism test which we describe in Section 3.1, and output a description of the graph modification in addition to the found sequences.

We also provide a Vertex Cover solver implementation with all our forward rules implemented, and some new data reduction rules which are explained later in this section. The solver is based on the branch-and-reduce paradigm and is very similar to the Vertex Cover solver by Akiba and Iwata [3]. Moreover, we provide a lifting algorithm that can transform solutions for the kernelized instances into solution for the original instances. We also provide a Python script that is used to visualize the graph modifications of the sequences of forward and backward rules that are output by our methods. However, our focus in this section is on the Find and Inflate-Deflate method.

Methodology. We have implemented our Find and Inflate-Deflate methods together with their two variations: Find and Reduce and local Inflate-Deflate. Almost all forward and backward rules described in the full version [10] are used by these methods, see Table 1 for an overview.

All rules increase or decrease k, but do not need to know k in advance. This allows us to run Find and Inflate-Deflate on all graphs, without having to specify a value for k. Instead, we set $k = 0$ for all instances. In the final instance (G', k') computed from $(G, k = 0)$ we will have $\tau(G') - k' = \tau(G)$, where $\tau(G)$ denotes the vertex cover number of G. For graphs G' which become empty, $-k'$ is the vertex cover number of the original graph G.

We set a maximum recursion limit for Find such that only sequences of at most two or three rules are found. We will refer to `FAR2` and `FAR3` as the Find and Reduce method which only searches for sequences of at most two or three forward and backward rules, respectively.

We used inflation ratios α equal to 10, 20 and 50 percent, and we will refer to the different Inflate-Deflate configurations as `ID10`, `ID20`, and `ID50`, respectively. We also tested our local Inflate-Deflate method with $\alpha = 20\%$, which we have found to work best in preliminary experiments, which we will refer to as `LID20`.

All these configurations were tested on the three datasets with a maximum running time of one hour.

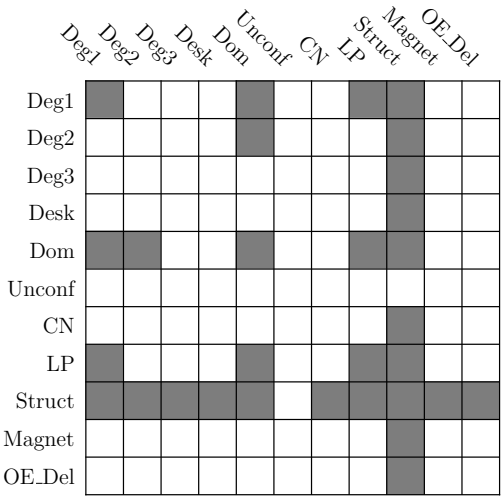


Figure 3 A matrix depicting which pairs of forward rules we have found to be non-confluent. A cell corresponding to a row rule R_a and a column rule R_b is colored white if the set $\mathcal{R} = \{R_a, R_b, \text{Deg0}\}$ is not confluent. The Degree-0 rule is always included to not take into account differences in number of isolated vertices left after applying the rules. Gray cells correspond to sets which may be confluent, meaning that no counter-example was found for them.

4.2 Results

Confluence. Confluence can be proved using for example conflict pair analysis [8], which is not straightforward and does not work for all types of data reduction rules. However, disproving confluence is potentially much easier, as it suffices to find one example where applying the rules in different order yields different instances.

For each pair of forward rules in Table 1, we tested on the set of all graphs with at most 9 vertices[3] whether we can obtain irreducible, but non-isomorphic graphs by randomly applying the two rules to the same graph. More precisely, we test whether a set $\mathcal{R} = \{R_a, R_b, \text{Deg0}\}$ is confluent for two forward rules R_a and R_b. We include the Degree-0 Rule in these sets, because a difference in the number of isolated vertices is only a minor detail which we do not wish to take into account. We note that, the inclusion of the Degree-0 Rule in these sets was never the reason that a set was not confluent.

Our results are summarized in Figure 3. The figure clearly shows that most pairs of forward rules are not confluent. This means that the relative order of these rules may affect the final instance.

Instance Shrinking. Next, we demonstrate how much the Find and Reduce and Inflate-Deflate methods were able to shrink the irreducible DIMACS, SNAP, and PACE graphs, which were obtained by exhaustively applying a set of forward rules. The results are summarized in Figure 4 and Table 2.

Notably, ten DIMACS and four SNAP instances were reduced to an empty graph by the Inflate-Deflate methods. Five further DIMACS graphs shrank to around 80% of their size, and half of the DIMACS graphs did not really shrink at all. We conclude that the Inflate-Deflate approach seems to either work really well or nearly not at all for a given instance.

[3] We obtained these graphs from Brendan McKay's website `https://users.cecs.anu.edu.au/~bdm/data/graphs.html`

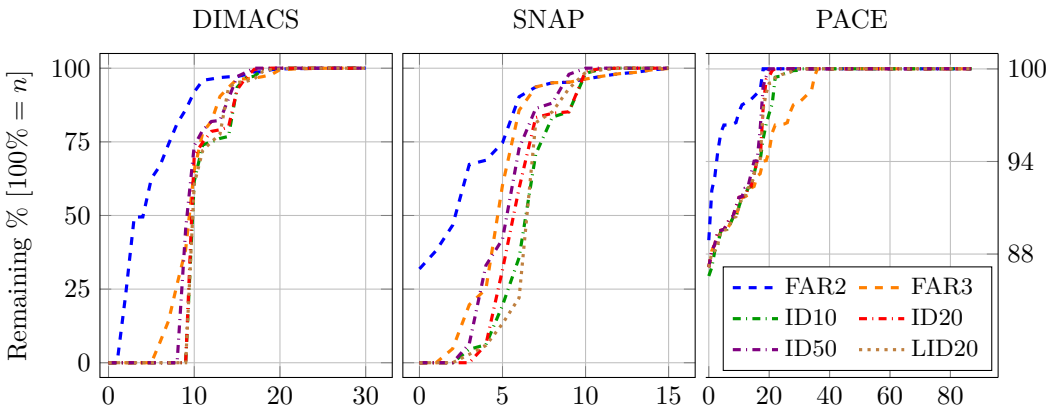


Figure 4 Cactus plots depicting how many irreducible instances were shrunk to a given fraction of their size (measured in vertices). The order of the instances is chosen for each configuration such that the size fractions of the instances are increasing. Note that the y-axis for DIMACS and SNAP start at zero (i. e. instances reduced to the empty graph); the y-axis for PACE does *not*.

Table 2 Summery of the average relative size (in terms of number of vertices) achieved by each of the configurations on the three datasets. The best value per dataset is given in bold.

	FAR2	FAR3	ID10	ID20	ID50	LID20
DIMACS	82.6%	67.0%	**62.9%**	63.6%	66.1%	63.4%
SNAP	80.6%	66.8%	**56.4%**	59.3%	64.1%	56.4%
PACE	99.2%	**97.4%**	97.8%	98.1%	98.1%	98.0%

On average the ID10 configuration produced the smallest irreducible instances, see Table 2. From the FAR2 configuration it can be seen that already applying only two forward/backward rules in a sequence can considerably reduce the size of some graphs. In this case, the first rule is always a backward rule, and the second always a forward rule. However, using up to three rules in a sequence gave significantly better results.

For Inflate-Deflate, we see that small inflation ratios α around 10% perform best on average. Increasing the inflation ratio leads to slightly worse results, especially for the SNAP instances.

Next, in Figure 5, we see that the ID10 method was able to shrink graphs to empty graphs mostly for graphs with the lowest average degree (8–14), with the exception of one instance with an average degree of around 45. However, a large number of graphs with an average degree of 8–14 were not shrunk considerably. The other configurations, namely ID20, ID50, and FAR3 exhibit the same behavior. Similar behavior is also observed by replacing the average degree with the maximum degree. For the most part, only graphs with a relatively small maximum degree were able to be shrunk considerably. We conclude that our approach is most viable on sparse irreducible graphs.

In Figure 6, we show how the graph size changes over time with Find and Reduce and Inflate-Deflate on two example graphs. It can be clearly observed how ID10 repeatedly increases the number of vertices, which is the inflation phase, and then subsequently reduces it, which is the deflation phase. A slow downward trend of the number of vertices can be observed.

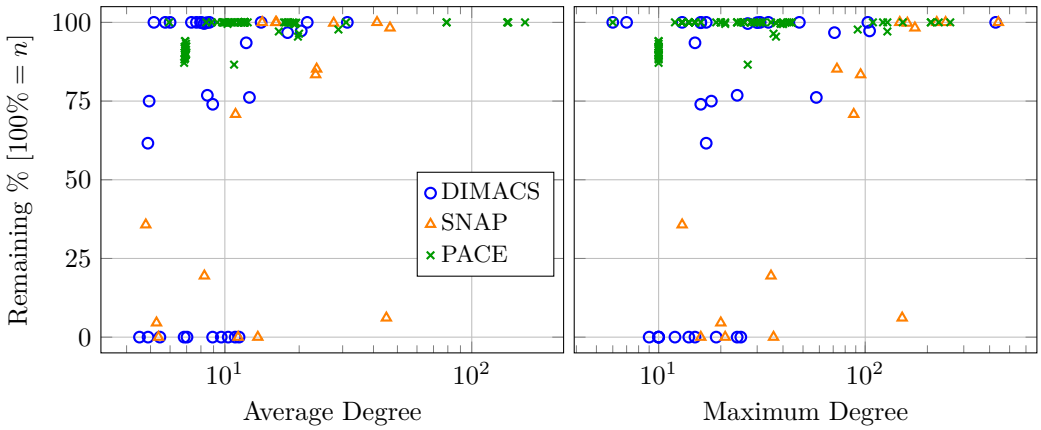


Figure 5 Scatter plot relating the relative size (measured in vertices) of the shrunk instances to the average and maximum degree of these graphs for the `ID10` configuration. Note that a logarithmic scale is used for the x-axis.

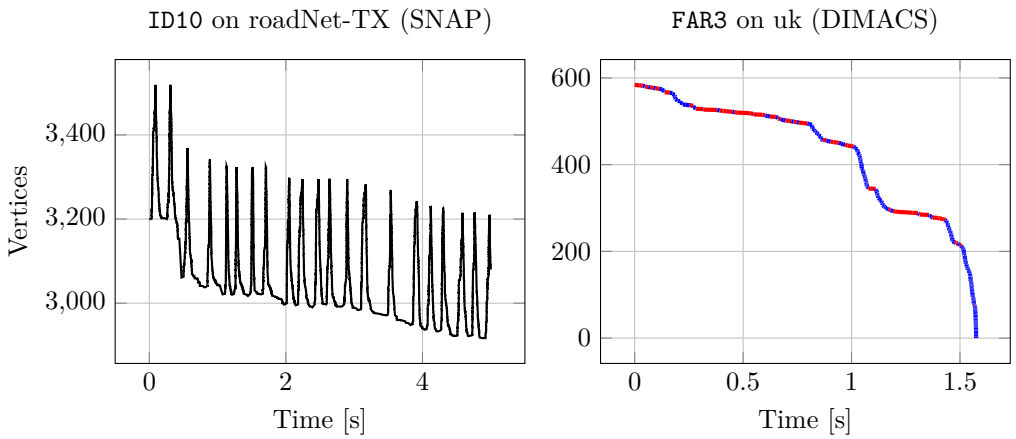


Figure 6 The shrinkage of two instances by ID10 and FAR3 configurations on already kernelized instances. The x axis is time, and y is number of vertices. In the right figure red segments mean that the graph was shrunk by using a sequence of at least two forward and backward rules, whereas the blue segments indicate the use of forward rules only.

For the `FAR3` configuration it can be seen that there are phases in which only forward rules have to be used to shrink the graph, and phases where longer sequences of forward and backward rules are needed. Sometimes a sudden large decrease in the number of vertices using only forward rules can be observed, which one may think of as a cascading effect. The graph only had to be changed by a small amount, triggering a cascade of forward rules.

5 Conclusion

Our work showed the large potential in the general idea of undoing data reduction rules to further shrink instances that are irreducible with respect to these rules. While the results for some instances are very promising, our experiments also revealed that other instances resist our attempts of shrinking them through preprocessing. From a theory point of view this is no surprise, as we deal with NP-hard problems after all. However, there is a lot of work

that still can be done in this direction. Similar to the branch&bound approach, many clever heuristic tricks will be needed to find solutions in the vast search space. Such heuristics might, for example, employ machine learning to guide the search. As mentioned before, our approach is not limited to Vertex Cover. Looking at other problems is future work though. A framework for applying our approach on graph problems could be another next step. Also, our approach should be easily parallelizable.

Besides all these practical questions, there are also clear theoretical challenges: For example, for a given set of data reduction rules is there *always* (for each possible instance) a sequence of backwards and forward rules to obtain an equivalent instance of constant size? Note that this would not contradict the NP-hardness of the problems: Such sequences are probably hard to find and could even be of exponential length.

References

1 Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Christopher T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments and the 1st Workshop on Analytic Algorithmics and Combinatorics*, pages 62–69. SIAM, 2004.

2 Faisal N. Abu-Khzam, Sebastian Lamm, Matthias Mnich, Alexander Noe, Christian Schulz, and Darren Strash. Recent advances in practical data reduction. *CoRR*, abs/2012.12594, 2020. arXiv:2012.12594.

3 Takuya Akiba and Yoichi Iwata. Branch-and-reduce exponential/fpt algorithms in practice: A case study of vertex cover. *Theoretical Computer Science*, 609:211–225, 2016. doi:10.1016/j.tcs.2015.09.023.

4 Gabriela Alexe, Peter L. Hammer, Vadim V. Lozin, and Dominique de Werra. Struction revisited. *Discrete Applied Mathematics*, 132(1-3):27–46, 2003. doi:10.1016/S0166-218X(03)00388-3.

5 David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013. doi:10.1090/conm/588.

6 M. Ayaz Dzulfikar, Johannes K. Fichte, and Markus Hecher. Pace2019: Track 1 - vertex cover instances, July 2019. doi:10.5281/zenodo.3368306.

7 M. Ayaz Dzulfikar, Johannes Klaus Fichte, and Markus Hecher. The PACE 2019 parameterized algorithms and computational experiments challenge: The fourth iteration (invited paper). In *Proccedings of the 14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*, volume 148 of *LIPIcs*, pages 25:1–25:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.IPEC.2019.25.

8 Hartmut Ehrig, Claudia Ermel, Falk Hüffner, Rolf Niedermeier, and Olga Runge. Confluence in data reduction: Bridging graph transformation and kernelization. *Computability*, 2(1):31–49, 2013. doi:10.3233/COM-13016.

9 Michael R. Fellows, Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller. What is known about vertex cover kernelization? In *Adventures Between Lower Bounds and Higher Altitudes - Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, volume 11011 of *LNCS*, pages 330–356. Springer, 2018. doi:10.1007/978-3-319-98355-4_19.

10 Aleksander Figiel, Vincent Froese, André Nichterlein, and Rolf Niedermeier. There and back again: On applying data reduction rules by undoing others, 2022. doi:10.48550/ARXIV.2206.14698.

11 Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, 2019. doi:10.1017/9781107415157.

12 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

13 Alexander Gellner, Sebastian Lamm, Christian Schulz, Darren Strash, and Bogdán Zaválnij. Boosting data reduction for the maximum weight independent set problem using increasing transformations. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX 2021)*, pages 128–142. SIAM, 2021. doi:10.1137/1.9781611976472.10.

14 Demian Hespe, Sebastian Lamm, Christian Schulz, and Darren Strash. WeGotYouCovered: The Winning Solver from the PACE 2019 Challenge, Vertex Cover Track. In *Proceedings of the SIAM Workshop on Combinatorial Scientific Computing, CSC 2020*, pages 1–11. SIAM, 2020. doi:10.1137/1.9781611976229.1.

15 Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, 1972. doi:10.1007/978-1-4684-2001-2_9.

16 Tomohiro Koana, Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. *ACM Journal of Experimental Algorithmics*, 26, 2021. doi:10.1145/3439801.

17 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. https://snap.stanford.edu/data, June 2014.

18 Karsten Weihe. Covering trains by stations or the power of data reduction. In *Proceedings 1st Conference on Algorithms and Experiments (ALEX98)*, pages 1–8, February 1998.