PaceYourself: Heuristic and Exact Solvers for the Minimum Dominating Set Problem

Lukas Geis
Goethe University Frankfurt, Germany

Alexander Leonhardt
Goethe University Frankfurt, Germany

Johannes Meintrup
THM, University of Applied Sciences Mittelhessen, Gießen, Germany

Ulrich Meyer
Goethe University Frankfurt, Germany

Manuel Penschuck
Goethe University Frankfurt, Germany

Abstract

Minimum-Dominating Set is a classical NP-complete problem. Given graph G, it asks to compute a smallest subset of nodes $\mathcal{D} \subseteq V(G)$ such that each node of G has at least one neighbor in $\mathcal{D}$ or is in $\mathcal{D}$ itself.

We submit two solvers to the PACE 2025 challenge, one to the exact track and one to the heuristic track. Both algorithms rely on heavy preprocessing with —to the best of our knowledge— novel reduction rules for the Dominating Set problem. The exact solver utilizes a reduction to the MaxSat problem to correctly identify a dominating set of minimum cardinality. The heuristic solver uses a randomized greedy local search to iteratively improve upon an initial dominating set as fast as possible.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms → Graph algorithms analysis

Keywords and phrases Dominating Set, Reduction Rule, Data Reduction, Practical Algorithm

Acknowledgements We would like to sincerely thank M. Grobler, S.Siebertz, and everyone else involved for their efforts in organizing PACE2025.

1 Introduction

In this document we describe an exact and a heuristic solver for the Minimum-Dominating Set. Both share the preprocessing phase outline in Section 3. It uses only safe data reduction rules to shrink the input instances, i.e., rules that allow us to recover the cardinality of an optimal solution. To the best of our knowledge, most of these data reduction rules were not described before — at least not in the context of Minimum-Dominating Set.

After preprocessing, our exact solver translates the instance into a MaxSat formulation that is handed over to external solvers (see Section 4). As discussed in Section 5, our heuristic uses repeated runs of a greedy search (using two different scoring functions) with randomized tie-braking for bootstrapping. It then relies on a carefully engineered local search scheme to optimize these initial solutions.

2 Preliminaries and Notation

Let $G = (V, E)$ be an undirected graph with $n := |V|$ nodes and $m := |E|$ (unweighted) edges. We denote the open neighborhood of a node $u \in V$ with $N(u) := \{v \in V \mid \{u, v\} \in E, u \neq v\}$ and the closed neighborhood of u with $N[u] = N(u) \cup \{u\}$. We define the degree $\deg(u) =$

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:8
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$|N(u)|$ of a node $u \in V$ as the number of (open) neighbors. For some $X \subseteq V$, we use $G[X] = (X, E_X)$ to denote the vertex-induced subgraph of $G = (V, E)$ where $E_X = \{\{u,v\} \in E \mid u, v \in X\}$.

The Minimum-Dominating Set asks to find a subset $D \subseteq V$ that is as small as possible, such that for every node $u \in V$, we have $N[u] \cap D \neq \emptyset$. Furthermore, let $V = \mathcal{U} \cup \mathcal{M}$ be a partition into the set of nodes $\mathcal{U}$ that have exactly one neighboring node in $\mathcal{D}$ in their closed neighborhood, and all remaining nodes $\mathcal{M}$. We define $N_{\mathcal{U}}[u] = N[u] \cap \mathcal{U}$, as the uniquely covered neighbors of u. If $u \in \mathcal{D}$ we say the nodes $N_{\mathcal{U}}[u]$ are uniquely covered by u.

3 Internal representation and preprocessing

Before running the main algorithms, we first attempt to reduce the size of the input graph G. To this end, we apply a multitude of reduction rules that may (i) modify the instance itself (delete nodes or edges) and (ii) assign nodes to the following (possibly overlapping) classes:

- **Selected nodes** $\mathcal{D}$ will become part of the solution set (i.e., there is an optimal dominating set including these nodes)
- **Covered nodes** $\mathcal{C}$ have at least one node in their closed neighborhood in $\mathcal{D}$ (this implies that $\mathcal{D} \subseteq \mathcal{C}$). Roughly speaking, nodes in $\mathcal{C}$ do not impose constraints, but may be useful to cover their neighbors.
- **Redundant[1] nodes** $\mathcal{R}$ are conceptually the opposite of covered nodes: a node $u \in \mathcal{R}$ may not be added into the solution $\mathcal{D}$, and thus requires at least one of its open neighbors to be selected. Observe that this class introduces additional constraints to reduce the search space by identifying "superfluous" nodes: To add a node u into $\mathcal{R}$, we have to proof that there exists a Minimum-Dominating Set $\mathcal{D}'$ that does not contain $\mathcal{R} \cup \{u\}$.
- As shortcuts, we define the complements $\overline{\mathcal{D}} = V \setminus \mathcal{D}$, as well as $\overline{\mathcal{C}} = V \setminus \mathcal{C}$, and $\overline{\mathcal{R}} = V \setminus \mathcal{R}$.

Thus, we can fully describe some intermediate state by $(G', \mathcal{D}, \mathcal{C}, \mathcal{R})$, where G' is the modified graph.[2] All our rules operate on this tuple. Before the first application, we initialize it as $(G, \mathcal{D}_0, \mathcal{D}_0, \emptyset)$, where G is the input graph and $\mathcal{D}_0 = \{u \in V \mid \deg(u) = 0\}$ the set of isolated vertices. After this point, all isolated nodes can be ignored.

Identifying *redundant nodes* $\mathcal{R}$ often boils down to a simple exchange-argument in which a neighbor is always at least as good as the redundant node itself. For example, consider two nodes $u, v \in V$, $u \neq v$ with $N[u] \subseteq N[v]$. Then, the only 'benefit' of adding u into the dominating set $\mathcal{D}$ is to cover nodes in $N[u]$. But because $N[v]$ is a superset of $N[u]$, adding v *instead* of u never yields a worse solution. Hence, we say that u is *subset-dominated* by v and can thus be marked as *redundant* (if v is not already marked as *redundant*).

We maintain the invariant that a classification cannot be undone, i.e., we may only add new nodes into the aforementioned sets $\mathcal{D}$, $\mathcal{C}$, and $\mathcal{R}$, but never delete existing ones. Since our rules are often applied iteratively, some care must be taken to uphold this invariant. For example, we need appropriate tie-breaking in the aforementioned *subset-domination* case to ensure that u and v do not change roles — even if they become twins (i.e., $N[u] = N[v]$) in later stages of the reductions.

This monotonic invariant is a quite important design decision in our solver, as it prevents "destructive interference" between rules. For instance, it generally is not possible to gleam

[1] The solver implementation refers to *redundant* nodes as *NeverSelect*.

[2] The `LongPaths` rule introduces a gadget, which requires additional post-processing. It is the only exception to this claim.

from $(G', \mathcal{D}, \mathcal{C}, \mathcal{R})$, *why* some previous decision was correct. Yet if we uphold the monotonicity and show that each rule is safe on it own, the overall safety follows inductively.

Trivial pruning based on node classes

The node classifications are often sufficient to shrink the graph. The key idea is that only *non-covered* nodes $u \in \overline{\mathcal{C}}$ can act as 'witnesses' to put a neighbor $v \in N[u]$ into $\mathcal{D}$. Similarly, only *non-redundant* nodes $v \in \overline{\mathcal{R}}$ are eligible to be put in $\mathcal{D}$ in the first place. Then consider a node $u \in V$:

- If $u \in \mathcal{C} \wedge u \in \mathcal{R}$: Since the node is redundant, it must never be added to $\mathcal{D}$. As it is already covered, it will also never act as a witness to select one its neighbors. Thus, we can safely remove u and all its incident edges from G.
- If $u \in \mathcal{C} \wedge u \in \overline{\mathcal{R}}$, the node is covered. But as it is not classified as redundant, it might still be put u into $\mathcal{D}$ to cover a subset of neighbors in $N(u)$. However, if a neighbor $v \in N(u)$ is already *covered*, it will not act as a witness for u and the edge $\{u, v\}$ can thus be safely deleted from G.
- If $u \in \overline{\mathcal{C}} \wedge u \in \mathcal{R}$, the redundant node u can still act as witness for one of it neighbors $N(u)$ — but only for non-redundant neighbors $N(u) \setminus \mathcal{R}$. Hence, if $v \in N(u)$ is also marked as *redundant*, the edge $\{u, v\}$ can be safely deleted.

We run this deletion-scheme after every application of every rule. Thus, we always assume that the input provided to a rule contains no edges between a pair of redundant nodes, no edges between a pair of covered nodes, and that all nodes that are covered and redundant have degree 0. At the same time, most reduction rules are phrased (and implemented) only in terms of adding nodes to classes; while implying the deletions.

We applied the following reduction rules exhaustively:

`CoveredLeaf.` If a node u is *covered* and has at most 1 *non-covered* neighbor $v \in N(u)$, mark u as redundant ($\mathcal{R} \leftarrow \mathcal{R} \cup \{u\}$) — this implicitly deletes u and $\{u, v\}$ from G. This rule is safe, since the only benefit of taking u into $\mathcal{D}$ is to cover v which can also be achieved by v (or any other neighbor of v). It is also the only rule that is part of the deletion-scheme itself and is thus run after every application of every other rule. In the special case that $v \in \mathcal{R}$ and $N(v) = \{u\}$, add u to $\mathcal{D}$ instead and mark v as *covered* — also deleting $\{u, v\}$ from G.

`SubsetRule.` This rule classifies nodes as *redundant* by the aforementioned subset-domination property. If $N[u] \subseteq N[v]$, then mark u as *redundant*. In case of a tie, break in favor of the node with higher index. We extend this notion by observing that only neighbors that are not already marked as *covered* are relevant for this property. Let $N_{\mathcal{C}}[u] = N[u] \setminus \mathcal{C}$ denote the subset of the closed neighborhood of u that is not *covered* yet. If $N_{\mathcal{C}}[u] \subseteq N_{\mathcal{C}}[v]$, mark u as *redundant* since the subset of potential witnesses for v is a superset of the set of potential witnesses for u.

`RuleOne.` For a node u, partition its neighborhood $N(u)$ into three distinct sets:

- $N_1(u) \coloneqq \{v \in N(u) \mid N(v) \setminus N[u] \neq \emptyset\}$,
- $N_2(u) \coloneqq \{v \in N(u) \setminus N_1(u) \mid N(v) \cap N_1(u) \neq \emptyset\}$,
- $N_3(u) \coloneqq N(u) \setminus N_1(u) \setminus N_2(u)$.

Alber et al. show in [1] that if $|N_3(u)| > 0$, it is optimal to put u into $\mathcal{D}$ and delete $N_2(u) \cup N_3(u)$ from the graph — replacing it with a single gadget leaf node. In our framework, we instead set $\mathcal{C} \leftarrow \mathcal{C} \cup N[u]$ and $\mathcal{D} \leftarrow \mathcal{D} \cup \{u\}$. We use a novel linear-time implementation of this rule that we describe and engineer in detail in [3].

Using ideas of `SubsetRule`, we further alter the original definition by putting every $v \in N_1(u)$ with $N(v) \setminus N[u] \subseteq \mathcal{C}$ into $N_2(u)$ instead. This is correct as u *subset-dominates* v which is the criterion for nodes in $N_2(u)$.

`SubsetRuleTwo.` Alber et al. extend `RuleOne` to pairs of nodes in a rule they dub RuleTwo [1]. For $u, v \in V, u \neq v$, we define $N(u,v) = N(u) \cup N(v)$ and $N[u,v] = N[u] \cup N[v]$:

- $N_1(u,v) \coloneqq \{x \in N(u,v) \mid N(x) \setminus N[u,v] \neq \emptyset\}$,
- $N_2(u,v) \coloneqq \{x \in N(u,v) \setminus N_1(u,v) \mid N(x) \cap N_1(u,v) \neq \emptyset\}$,
- $N_3(u,v) \coloneqq N(u,v) \setminus N_1(u,v) \setminus N_2(u,v)$.

If $|N_3(u,v)| > 1$ **and** no node in $N_2(u,v) \cup N_3(u,v)$ is incident to every node in $N_3(u,v)$, one can either add u and/or v to $\mathcal{D}$ and/or mark every node in $N_2(u,v) \cup N_3(u,v)$ as *redundant*. As the original rule is — even with optimizations of [3] — prohibitively slow on bigger instances, we restrict ourselves to a subset of RuleTwo in which every node $x \in N_2(u,v) \cup N_3(u,v)$ is either *subset-dominated* by u or v, or connected to both u and v. We also apply similar changes as in `RuleOne` for classification of nodes in $N_2(u,v)$.

`RedundantTwins.` `SubsetRule` and `SubsetRuleTwo` lead to many *redundant* nodes $\mathcal{R}$. After deleting all edges between redundant endpoints, redundant nodes can become twins (this happens quite often in the PACE dataset). Since a single witness suffices, all but one node of each set of twins can be removed.

`Isolated.` If every neighbor $N(u)$ of some node $u \in \overline{\mathcal{C}}$ is marked as *redundant*, we add u to the solution $\mathcal{D}$. Thereby we also cover all neighbors, which implies their deletion.

`RedundantCover.` Consider a "redundant triangle" on pairwise different nodes $r, u, v \in V$ where node $r \in \mathcal{R}$; as we remove all edges between redundant nodes, we know that $u, v \in \overline{\mathcal{R}}$. Since node r must not be added to the solution, we further know that at least u or v will become part of the solution and then cover the other two. Thus, u and v do not benefit from neighbors $w \in N(u) \cup N(v)$ that may provide coverage for them. This allows us to delete all edges $\{u, w\}$ to covered neighbors $w \in N(u) \cap \mathcal{C}$ (and analogously for v).

`VertexCover.` Consider a "redundant triangle" on pairwise different nodes $r, u, v \in V$ where node $r \in \mathcal{R}$ (see rule `RedundantCover`). Since either u or v need to be added to the solution, we can interpret it as a (trivial) vertex cover problem on the baseline edge $\{u, v\}$. Based on this observation, we conceptually compute a "vertex cover graph" G_{VC} consisting of all baseline edges of redundant triangles.

Now we solve vertex cover on special structures in G_{VC}; more specifically, the only structure which we identified sufficiently frequent are cliques. Observe that the vertex cover of any complete graph K_n consists of $n-1$ nodes. Thus, we search for a (maximal) clique C in G_{VC} which has at least one "internal" node $u \in C$, s.t. all neighbors are either in the clique or part of redundant triangles that formed the clique. Then, we assign $C \setminus \{u\}$ to the solution covering all neighbors $N[C \setminus \{u\}]$. This implicitly deletes C and all its redundant triangles.

`SmallExact.` We may compute a Minimum-Dominating Set as the union of optimal solutions for each connected component. Even if the input is connected, previous reduction rules may delete sufficiently many edges and nodes to disconnect parts of the graph. At the same time, small connected components can be dealt with generic solvers for mixed integer linear programs (MILP). Thus, after all other rules have been exhausted, we search for small connected components. For each small component, we construct an ILP formulation and attempt to solve it using HiGHS [5] with a very short timeout. To reduce overheads, we combine sufficiently small components into a single ILP problem.

The ILP is constructed in the straight-forward manner (for simplicity we formulate it for the whole graph; restriction to subgraphs is trivial): Each non-redundant node $u \in \overline{\mathcal{R}}$ corresponds to a binary variable x_u and we want to minimize their sum $\sum_u x_u$. Each uncovered node $u \in \overline{\mathcal{C}}$ adds the constraint $\sum_{v \in (N[u] \setminus \mathcal{R})} x_v \geq 1$. As an optimization, we can drop the following constraints: Consider an induced triangle on the three different nodes $r, u, v \in V$ where $r \in \mathcal{R}$. Thus, node r forces at least u or v into the solution; the edge $\{u, v\}$ ensures that either will cover the other. Hence, we can omit the constraints of u and v (which may have high degree!) in favor of the simple constraint $x_u + x_v \geq 1$.

`ArticulationPoint.` An articulation point $u \in V$ is a cut-vertex, whose removal disconnects a component. The set $A \subseteq V$ of all articulation points in a graph can be computed in linear time. [4] For each node $a \in A$, we test whether its removal results in at least one small connected components $C \subseteq V$. Then, we attempt to solve the subproblem $G[C']$ induced by $C' = C \cup \{a\}$ using the ILP formulation discussed for rule `SmallExact`.

There is one complication: by restricting to C', the ILP does not encode the full context anymore. Without this, we cannot properly decide whether in a globally optimal solution (i) node a covers itself, and/or whether a node (ii) in C, or (iii) in $V \setminus C'$ takes over this role.

Suppose that all optimal global solutions cover a only from the outside (i.e., case iii). Then, requiring the $G[C']$ to cover a "from within" leads to suboptimal solutions. To prevent this case, we treat a as already being covered while solving the ILP.

This of course leads to issues, if globally optimal solutions do, in fact, require a to be covered from within C'. Then there are two cases: either there exists a minimum-Dominating Set on $G[C']$ that includes node a. Otherwise, adding a will increase the solution size by one. Thus, we setup a weighted variant of the ILP that is biased towards nodes near a; formally, the cost function to minimize becomes $\sum_u \alpha_u x_u$, where

$$\alpha_u = \begin{cases} 1 - 2\varepsilon & \text{if } u = a \\ 1 - \varepsilon & \text{if } u \in N(a) \\ 1 & \text{otherwise} \end{cases} . \tag{1}$$

For $0 < \varepsilon < 1/(2|C'|)$, this will select a Minimum-Dominating Set on C' and favor those that include a, or (with smaller priority) a neighbor of a. It will, however, never increase the solution size on $G[C']$.

`LongPaths.` The long path rule searches for induced paths $P = (s, u_1, u_2, \ldots, u_k, t)$ in G where $\deg(u_i) = 2, \forall i \colon 1 \leq i \leq k$. We implement various special cases if $s = t$ (i.e., P is a cycle) or either one or both endpoints e_i are leafs. These are already implied by `RuleOne`, `SmallExact`, or `ArticulationPoint` but can be more efficiently addressed here. However, since correctness follows from these rules, we omit a detailed discussion here.

The remaining case is $s \neq t \ \wedge \ \deg(s) > 2 \ \wedge \ deg(t) > 2$. As soon as any of the nodes in P is covered or redundant, we can optimally solve the path in a single scan. Otherwise if all nodes are unclassified and $k \geq 5$, we can shorten the path. In this case, we delete the nodes $u_2, \ldots, u_{1+3\ell}$ (where $\ell \in \mathbb{N}$) and instead add the edge $\{u_1, u_{3\ell+2}\}$. We record the removed edges. After the solver computed a solution on the reduced graph, a post-processing reintroduces the removed edges and solves them in a single scan based on the solved context.

4 Exact Solver

Our exact solver is explicitly designed to test the effectiveness of our reduction rules when preprocessing inputs for *unmodified off-the-shelve* solvers. We consider this an interesting line of inquiry, since general-purpose solvers integrate extensive advancements in solving broad optimization problems, whereas problem-specific preprocessing can significantly leverage domain-specific knowledge to enhance performance.

To this end, we conducted experiments with several ILP solvers (including HiGHS, gurobi, coin-cbc, scip) and MaxSat solvers (most submissions of the MaxSAT 2024[3] competition). Ultimately, two different MaxSat solvers were selected since their performance characteristics complement quite nicely: after preprocessing, we first run UWrMaxSat[4] by M. Piotrów with a timeout of 600s; if no solution was found within the time budget, we start EvalMaxSAT[5] by F. Avellaneda.

Both solvers support the concept of soft and hard constraints, where all hard constraints have to be satisfied while minimizing the number of violated soft constraints. Similarly to the ILP formulation discussed earlier, each non-redundant nodes is assigned a binary predicate x_u; where node $u \in V$ is part of the solution $\mathcal{D}$ iff $x_u = 1$. Each non-covered neighbor then emits a hard constraint that at least one node in its closed neighbors must be included. In order to minimize the number of selected nodes, we produce a soft constraint $\neg x_u$ for each predicate x_u.

5 Heuristic Solver

The strategy of our heuristic solver is based on a local search heuristic, which has been shown to work well for finding minimum dominating sets [8], and a wide variety of other NP-complete problems [2, 6]. Before running the search however, we remap and relabel $(G, \mathcal{D}, \mathcal{C}, \mathcal{R})$ to the induced subgraph $(G', \mathcal{D}', \mathcal{C}', \mathcal{R}')$ that does not contain isolated vertices. As each node in $\mathcal{D}$ is isolated after our deletion scheme, the induced subgraph has no nodes in $\mathcal{D}'$ at the start. After running the local search, we map the resulting $\mathcal{D}'$ back to the original graph concatenating it with the preprocessed $\mathcal{D}$ to obtain a valid dominating set for G.

In each iteration of the local search process the heuristic solver chooses between one of two possible actions:

Eviction *(rarely)*. Evict a single node v from the dominating set $\mathcal{D}'$ to form $\mathcal{D}'_t = \mathcal{D}' \setminus \{v\}$. In the following we greedily add nodes to $\mathcal{D}'_t$, while avoiding v, until $\mathcal{D}'_t$ is a valid dominating set again.

Swap *(frequently)*. Pick a vertex $v \in \overline{\mathcal{D}'}$ for which there exists a $(x, 1)$-swap for $x \geq 1$. A $(x, 1)$-swap creates a new valid dominating set $\mathcal{D}'_t = (\mathcal{D}' \setminus \{v_1, v_2, \ldots, v_x\}) \cup \{v\}$ by the addition of a single new vertex and the removal of x former constituents of $\mathcal{D}'$.

As opposed to the local search by Zhu et al. [8], we maintain the invariant that at the end of each round the ensuing dominating set $\mathcal{D}'$ is valid. This is an important design choice, as it confers some algorithmic benefits while having mixed effects on the traversal of the solution space by the local search procedure. On one hand it constrains the new solutions that can be possibly reached by one of the aforementioned actions. On the other hand it implies that while searching for a better solution we always stay close to an actual solution instead of (possibly) straying arbitrarily far from any valid solution. But most importantly, as stated before, the *swap* action is the most prevalent one in our solver, and maintaining

[3] `https://maxsat-evaluations.github.io/2024/`.
[4] `https://maxsat-evaluations.github.io/2024/mse24-solver-src/exact/unweighted/UWrMaxSat-SCIP-MaxPre.zip` based on [7]
[5] `https://maxsat-evaluations.github.io/2024/mse24-solver-src/exact/unweighted/EvalMaxSAT_2024.zip`

the previously mentioned invariant allows for an efficient datastructure to maintain a set of eligible canidates for it.

Throughout the local search procedure, we dynamically maintain a tree $\mathcal{T}_v$ for each node $v \in D$ that keeps track of the intersection of the closed neighborhoods of all nodes in $N_{\mathcal{U}}[v]$. Recall that $N_{\mathcal{U}}[v]$ are the neighbors of v that are adjacent to exactly one node in $\mathcal{D}'$. Since $v \in D$ this implies v is the one and only node in the dominating set adjacent to these neighbors. Clearly, there exists an $(x, 1)$-swap if there is a set $S = \{v_1, \ldots, v_k\} \subseteq \mathcal{D}'$ and a vertex $u \in \overline{\mathcal{D}'}$ such that

$$\bigcup_{1 \leq i \leq k} N_{\mathcal{U}}[v_i] \subseteq N[u] \tag{2}$$

where $1 \leq x \leq k$. Observe, that the previous condition is necessary but not sufficient to establish $x = k$ due to overlapping neighborhoods.[6] Therefore, if we dynamically maintain the tree $\mathcal{T}_v$ with vertex set $N_{\mathcal{U}}[v]$ where each inner node $u \in N_{\mathcal{U}}[v]$ of the tree is the intersection of the closed neighborhoods of all nodes in the subtree rooted in u, we can make several observations:

1. The root of $\mathcal{T}_v$ contains all nodes that are eligible for a $(1, 1)$-swap where v is swapped out of the dominating set.
2. We can maintain this datastructure in $\mathcal{O}(m)$[7] space and $\mathcal{O}(\Delta \log \Delta)$ time per update of $\mathcal{T}_v$ where Δ is the maximum degree of the input graph.
3. If we maintain for all nodes $u \in \overline{\mathcal{D}'}$ a counter how often they appear in the root of some tree $\mathcal{T}_v$ we recover k for the condition mentioned in Equation (2).

By virtue of the previous observations we are able to use a random weighted sampling procedure where the weight of $u \in \overline{\mathcal{D}'}$ is given by $w_u = 2^k$ where k is the number of nodes in $\mathcal{D}'$ for which u is within the root of their respective trees.[8] Upon executing a swap we dynamically remove and add the former and newly *uniquely covered* neighbors to and from the trees of their respective unique coverer. To support this efficiently, it is essential for us to know the unique covering node when (i) a node that was covered by two nodes in $\mathcal{D}'$ is now uniquely covered, (ii) a node loses the property of being uniquely covered since another neighboring node entered $\mathcal{D}'$. We compactly represent the previously mentioned requirements by storing the covering nodes of any node $u \in V$ as the XOR'ed signature $\bigoplus_{v \in N[u] \cap D} v$ of the set of u covering nodes. Clearly, addition and removal are the same operation depending on the stored XOR'ed signature due to the commutativity of $\oplus$. If a node is *uniquely covered* the XOR'ed signature is exactly the covering node. This allows to store a large set of covering nodes cache-efficiently, while being able to retrieve the unique covering node at the aforementioned critical points in time.

Working set. After any swap we keep track of all nodes within the roots of all dominating set nodes whose uniquely covered neighbor sets were shrinked by the most recent swap. We preferentially sample multiple times from this *working set* and tie-break by considering the aforementioned score to enhance the locality of our heuristic.

Clearly, the *swap* action makes the solver prone to enter local minima, without any means to leave them again. Therefore, we evict a single vertex from $\mathcal{D}'$ either if there has been no

---

[6] Consider for example w, a node neighbored by only two nodes within $\mathcal{D}'$, say v_1 and v_2 and assume $w \notin N[u]$. Since $w \in \mathcal{M}$, the stated condition does not assert that u covers w as well, therefore u cannot replace both v_1 and v_2, but it can always replace at least one of them.

[7] For this it suffices to see that $\bigcup_{v \in D} N_{\mathcal{U}}[v]$ is always a partition of $\mathcal{U}$.

[8] For practical reasons we clamp k to 5.

improvement to the current solution for some time, or if the weighted sampling structure is empty. We rely on three different procedures each with equal probability when evicting a vertex (i) we randomly choose a vertex from $\mathcal{D}'$, (ii) we randomly choose a vertex $v \in \mathcal{D}'$ where the root of $\mathcal{T}_v$ only contains v and tie-break by frequency and age, (iii) we randomly choose a vertex $v \in \mathcal{D}'$ where root of $\mathcal{T}_v$ only contains v and tie-break by the cardinality of $|N_{\mathcal{U}}[u]|$ and age. Here, the frequency is defined as the number of times a vertex has left $\mathcal{D}'$ during the local search and the age is defined as the last iteration that a node has either entered of left $\mathcal{D}'$.

References

1. Jochen Alber, Michael R. Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *J. ACM*, 51(3):363–384, 2004.
2. Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. Numvc: An efficient local search algorithm for minimum vertex cover. *J. Artif. Intell. Res.*, 46:687–716, 2013.
3. Lukas Geis, Alexander Leonhardt, Johannes Meintrup, Ulrich Meyer, and Manuel Penschuck. Simpler, better, faster, stronger: Revisiting a successfull reduction rule for dominating set. 2025.
4. John E. Hopcroft and Robert Endre Tarjan. Efficient algorithms for graph manipulation [H] (algorithm 447). *Commun. ACM*, 16(6):372–378, 1973.
5. Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.
6. Nabil H. Mustafa and Saurabh Ray. PTAS for geometric hitting set problems via local search. In *SCG*, pages 17–22. ACM, 2009.
7. Marek Piotrów. Uwrmaxsat: Efficient solver for maxsat and pseudo-boolean problems. In *ICTAI*, pages 132–136. IEEE, 2020.
8. Enqiang Zhu, Yu Zhang, Shengzhi Wang, Darren Strash, and Chanjuan Liu. A dual-mode local search algorithm for solving the minimum dominating set problem. *Knowl. Based Syst.*, 298:111950, 2024.