PACE Solver Description: *Bad Dominating Set Maker*

Alexander Dobler
Algorithms and Complexity Group, TU Wien, Austria

Simon Dominik Fink
Algorithms and Complexity Group, TU Wien, Austria

Mathis Rocton
Algorithms and Complexity Group, TU Wien, Austria

Abstract

We present *Bad Dominating Set Maker*, our solver for the exact tracks of the PACE Challenge 2025 for both the Dominating Set and Hitting Set problems. It uses reduction rules, dynamic programming on tree decompositions, and external Vertex Cover and SAT solvers.

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

Keywords and phrases Dominating Set, Hitting Set, Pace Challenge

Digital Object Identifier 10.4230/LIPIcs...

Supplementary Material *Software (source code)*: `https://github.com/Doblalex/pace2025/tree/submission`

Funding *Alexander Dobler*: Supported by the Vienna Science and Technology Fund (WWTF) under grant [10.47379/ICT19035].
Simon Dominik Fink: Funded by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT22029]
Mathis Rocton: Supported by the European Union's Horizon 2020 research and innovation COFUND programme (LogiCS@TUWien, grant agreement No 101034440), and the FWF Science Fund (FWF project Y1329)

1 Introduction

Dominating Set and Hitting Set are two well-known NP-hard problems on graphs and hypergraphs respectively. In the Dominating Set case, one is looking for a subset S of vertices of minimum size, such that all vertices have a neighbor in S, whereas for Hitting Set we require that this minimu size subset S intersects each hyperedge: Hitting Set is the generalization of Vertex Cover to hypergraphs.

2 *Bad Dominating Set Maker*

Our algorithm can be broken down in five main steps. It first reads the input and constructs the corresponding instance of Directed Constrained Domination, which we define in Section 3. Note that only this step makes a difference between dominating set instances and hitting set instances. Once we have constructed the instance, we apply exhaustively a set of reduction rules described in Section 4. Some of these reduction rules are splitting the instances into several smaller, in which case we then solve these instances independently. Once the reduction rules are exhausted, we use the library *htd* [1, 2] to look for a tree decomposition of the undirected underlying graph with width smaller than 13; if we find such a decomposition we solve the problem using a dynamic programming, as described in

Section 5. Otherwise, if the instance corresponds to a Vertex Cover instance and has, informally, not too many candidates for the solution set, we run the dedicated solver *Peaty* [6] with a 5-minute time limit, see Section 6. Finally, in any other case we create a Max SAT instance equivalent to our current Directed Constrained Domination instance, and use the *EvalMaxSAT* solver [3, 4], with some improvements regarding the core computation described in Section 7.

To see that our solver is correct, it suffices to observe in the following sections that:

- the reduction rules are safe,
- the dynamic programming algorithm on tree decompositions is exact,
- *Peaty* and *EvalMaxSAT* are exact solvers,
- the conversions between the different problems –Hitting Set, Dominating Set, Directed Constrained Domination, Vertex Cover, SAT– are correct.

3 Directed Constrained Domination

We reduce both Dominating Set and Hitting Set instances to instances of the following problem: Directed Constrained Domination
Input: A directed graph $G = (V, A)$, a subsumed set $S \subseteq V$ and a dominated set $D \subseteq V$.
Goal: A set $DS \subseteq V \setminus S$ of minimum size, such that: $\forall v \in V \setminus D,\ \exists u \in DS, uv \in A$.

Note that A can contain arcs in both directions between two vertices, as well as self-loops. The transformation from Dominating Set to Directed Constrained Domination is trivial: the vertex set is the same and every edge is replaced by arcs in both directions, and no vertex is flagged as subsumed nor as dominated.

From Hitting Set, it suffices to create a dominated vertex for each element of the universe, and then a subsumed vertex for each set. Then, adding arcs from each dominated vertex to the subsumed vertices representing sets it belongs to in the Hitting Set instance suffices: any set will be hit if and only if the solution for Directed Constrained Domination contains a vertex belonging to this set.

4 Reduction Rules

Our algorithm first preprocesses the instances with a large set of reduction rules to simplify the instance and make it smaller, thus easier to handle for the exact solvers called later on. The reduction rules we use are adapted and/or generalized to our Directed Constrained Domination problem from the rules available for Dominating Set in the work of Van Rooij and Bodlaender [7] and Volkmann [9].

The following reduction rule always has to be applied to ensure the safeness of the other reduction rules.

▶ **Reduction Rule 1** (Arc removal). *If a vertex is dominated, remove all its in-arcs. If a vertex is subsumed, remove all its out-arcs.*

We adapt the classical reduction rules on vertices with extreme degrees and connected components to the directed setting. For the rules 5 and above, keep in mind that due to Reduction Rule 1, an uv arc implies that u is not subsumed and v is not dominated.

▶ **Reduction Rule 2** (In-degree 0). *If there is an undominated vertex v with in-degree 0, add v to the dominating set.*

▶ **Reduction Rule 3** (Out-degree 0). *If there is a dominated vertex v with out-degree 0, remove v from the instance.*

▶ **Reduction Rule 4** (Connected components). *Connected components can be solved independently.*

▶ **Reduction Rule 5** (In-degree 1). *Let v be a vertex with a single incoming arc uv, and no outgoing arcs except for (possibly) vu. Add u to the dominating set and remove v.*

▶ **Reduction Rule 6** (Out-degree 1). *Let v be a dominated vertex with a single outgoing arc vu. If u is not subsumed or has at least in-degree 2, remove v. Otherwise, add v to the dominating set.*

▶ **Reduction Rule 7** (Universal dominated vertex). *Let v be a vertex with in-degree $n-1$. If some other vertex is not yet dominated, mark v as dominated. Otherwise, add v to the dominating set and remove all vertices and arcs.*

▶ **Reduction Rule 8** (Universal dominating vertex). *Let v be a vertex with out-degree $n-1$. Add v to the dominating set and remove all vertices and arcs.*

The efficient implementation of the following rule required a *partition refinement* algorithm to compute the set-inclusion of neighborhoods. The success of this approach is due to the fact that we do not need to save the whole information at each step, but can "forget" what is either irrelevant (information about empty parts in the partition) or easily recoverable at the end, *i.e.* implied by the transitivity of the inclusion relation.

▶ **Reduction Rule 9** (Subsumption). *Let u, v be two vertices.*
If $N_{out}[u] \supseteq N_{out}[v]$, mark v as subsumed.
Independently, if $N_{in}[u] \subseteq N_{in}[v]$, mark v as dominated.

▶ **Reduction Rule 10** (Contraction). *Let u, v be two vertices.*
If u is dominated but not subsumed, and v is not dominated but subsumed, and there is an arc from u to v: mark u as not dominated and add to its in-neighborhood the in-neighborhood of v, and delete v.

Proof. The set of vertices that can be selected in the dominating set are the same before and after the update, since v is subsumed. Since v (resp. u) is not dominated in the initial graph (resp. in the updated graph), in any solution there will be at least a vertex x in its in-neighborhood. Remark that these in-neighborhoods are the same. Now, let us consider the graph remaining after selecting a such x and adding it to the dominating set. In both of the following cases, $x = u$ or $x \neq u$, both the initial and updated graph yield the exact same instance, after applying 3 to v in the initial instance. Hence, there exist a solution for the updated instance if and only if there exist one for the initial instance, and the reduction is safe.
Moreover, any optimal solution for one instance is also an optimal solution for the other, with no modification necessary. ◀

The following two rules, following principles identified in rules 6 and 7 of [7], required more care in translating to our setting than the previous rules. Indeed, our first special rule is a generalization of their rule number 6, considering that we do not have control over the maximal size of out-neighborhoods as they do, and thus we need to compute a very local dominating set instead of simply counting elements. For efficient implementation, we want to apply the second special reduction rule several times without applying the subsumption reduction rule between uses. Because of this, we need to make sure that some vertices do not have the same in-neighborhoods before reducing.

▶ **Reduction Rule 11** (Special 1). *Let u be a vertex and $N_2^+(u)$ be the out-neighbors of u having in-degree exactly 2. Let P be the union of the in-neighborhoods of $N_2^+(u)$. Let Q be the union of the out-neighborhoods of P, minus the out-neighborhood of u. Let s_Q be the minimum size of a set dominating Q.*
If $1 + s_Q \leq |P|$, take u in the solution.

Proof. Idea: If u is not taken in the solution, we need to take all of P in the solution to cover the out-neighbors of u having in-degree exactly 2. However, this would cost more and be less efficient than taking u and a minimum sized dominating set of Q. ◀

▶ **Reduction Rule 12** (Special 2). *Let v a vertex with exactly 2 out-neighbors u_1, u_2 having in-degree 2. We denote by w_1 and w_2 the remaining in-neighbors of u_1 and u_2, respectively. We proceed only if $w_1 \neq w_2$*
Mark u_1 and u_2 as dominated, create a new dominated vertex with out-neighborhood the union of the out-neighborhoods of w_1 and w_2, and subsume w_1 and w_2.
If the solution for this new instance contains w, replace it by w_1 and w_2 to obtain a solution for the initial instance, and otherwise add v to the solution set.

Proof. Idea: Observe that we need either v or the pair w_1, w_2 in the solution to dominate u_1 and u_2. If we need w in the reduced instance, we would need at least one of w_1, w_2 in the solution for the initial instance to cover what w covers, and taking the other of w_1, w_2 is then optimal, since it dominates more than what v can still dominate. In the other case, w is not necessary thus v is a smaller (better) dominating set for $\{u_1, u_2\}$. ◀

▶ **Reduction Rule 13** (Cut-vertices [9]). *Let v be an (undirected) cut-vertex separating the subgraphs $B_1, \ldots, B_k$ in G. For $i = 1, \ldots, k$, let B_i^- be B_i with v marked as dominated and B_i^+ be B_i with v added to the dominating set. The optimal dominating set $DS(G)$ is the smallest of $\bigcup_i DS(B_i^+) \cup \{v\}$ and $\bigcup_{i \neg j} DS(B_i^-) \cup DS(B_j)$ for $j = 1, \ldots, k$.*

5 Solving Low-Treewidth Instances

We use the *htd* library with the mindegree strategy in the hope of finding a nice tree-decomposition of small width [1, 2]. If we obtain such a decomposition, we use a dynamic programming algorithm running in time $\mathcal{O}(3^{\mathrm{tw}})$ to solve exactly the instance. To achieve this running time, our approach mixes together two different setups of the approach described in [8].
Namely, for join nodes, we consider tables where the state of the vertices are 1, $0_?$ and 0_0, whereas for introduce and forget nodes we consider the states 1, 0_1 and 0_0. 1 means the vertex is selected in the solution, 0_1 that it is not in the solution but neighbor to a vertex in the solution, 0_0 that it is neither in the solution nor has a neighbor in the solution. The additional $O_?$ state (which is introduced to make the table computation at join nodes faster) corresponds to a vertex not selected in the solution, without information about its neighborhood. Our algorithm translates then the tables from one format to another according to the type of node we encounter in the tree decomposition.
Additionally, we need to restrict the possible states for some vertices to adapt the algorithm to our needs: any subsumed vertex cannot be given state 1, and any dominated vertex can be given the state 0_1 (or $0_?$) for free.

6 Handling Vertex Cover Instances

We remark that some of the instances are encoding instances of Vertex Cover. For such instances with less than 1000 vertices remaining unsubsumed, we want to use a dedicated solver. We use the following observation: if all the vertices with incoming arcs (*i.e.* the vertices needing to be dominated) have in-degree exactly 2 and out-degree 0, the instance corresponds exactly to the instance of Vertex Cover where vertices with outgoing arcs are vertices, and vertices with ingoing arcs are edges between their two neighbors.

We then run for a maximum of 5 minutes the solver *Peaty* on the obtained Vertex Cover instance. *Peaty* is an exact solver having achieved second place in the PACE Challenge 2019 [5, 6].

7 EvalMaxSAT and Improvements

As the final step of our algorithm, we encode the remaining instance in a SAT formula, and use the *EvalMaxSAT* solver to tackle it [3, 4]. The encoding of the instance is trivial, as it suffices to encode each in-neighborhood of the graph with a clause ensuring that one of the vertices will be selected, and the selection of any vertex has a weight of -1, meaning that the solver is looking for a solution selecting as few vertices as possible.

To improve the performance in our setting, we provide two adjustments.

First, we use an adaptative time-out for the *core improvement* subroutine of the solver, to spend less time when we believe the lower bound to be already optimal.

The second improvement takes care of providing an alternative initial lower bound to the solver: we look for a large matching in the accessory graph where there is an edge between two vertices if this pair is exactly the in-neighborhood of some vertex in the Directed Constrained Domination instance. Since the size of such a matching is always a lower bound for the size of a solution, we can feed this lower bound to *EvalMaxSAT* if the one it obtains is not as good.

References

1 Michael Abseher, Nysret Musliu, and Stefan Woltran. *htd*: A small but efficient c++ library for computing (customized) tree and hypertree decompositions. https://github.com/mabseher/htd.

2 Michael Abseher, Nysret Musliu, and Stefan Woltran. htd–a free, open-source framework for (customized) tree decompositions and beyond. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 376–386. Springer, 2017.

3 Florent Avellaneda. Evalmaxsat. https://github.com/FlorentAvellaneda/EvalMaxSAT.

4 Florent Avellaneda. Evalmaxsat 2023. *MaxSAT Evaluation 2023*, page 12, 2023.

5 M Ayaz Dzulfikar, Johannes K Fichte, and Markus Hecher. The pace 2019 parameterized algorithms and computational experiments challenge: the fourth iteration. In *14th International Symposium on Parameterized and Exact Computation (IPEC 2019)*, pages 25–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.

6 Patrick Prosser and James Trimble. Peaty: an exact solver for the vertex cover problem. https://github.com/jamestrimble/peaty/tree/master.

7 Johan MM Van Rooij and Hans L Bodlaender. Exact algorithms for dominating set. *Discrete Applied Mathematics*, 159(17):2147–2164, 2011.

8 Johan MM van Rooij, Hans L Bodlaender, Erik Jan van Leeuwen, Peter Rossmanith, and Martin Vatshelle. Fast dynamic programming on graph decompositions. *arXiv preprint arXiv:1806.01667*, 2018.

9 Lutz Volkmann. A reduction principle concerning minimum dominating sets in graphs. *JOURNAL OF COMBINATORIAL MATHEMATICS AND COMBINATORIAL COMPUTING*, 31:85–90, 1999.