


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

DOMINATING SET and HITTING SET are two well-known NP-hard problems on graphs and hypergraphs, respectively. For DOMINATING SET, we seek a subset S of vertices of minimum size, such that every vertex has a neighbor in S . For HITTING SET, we require that this minimum size subset S intersects each hyperedge. We present *Bad Dominating Set Maker*, our solver for both problems posed in the exact tracks of the 2025 PACE Challenge. 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.IPEC.2025.35

Supplementary Material *Software (source code)*: <https://github.com/Doblaalex/pace2025> [3]
archived at `swb:1:dir:77b0720cd14b92e8d17e114ccf5c1ea89f79dfc9`

Funding *Alexander Dobler*: Vienna Science and Technology Fund (WWTF) grant [10.47379/ICT19035].

Simon Dominik Fink: Vienna Science and Technology Fund (WWTF) grant [10.47379/ICT22029] and Austrian Science Fund (FWF) grant [10.55776/Y1329].

Mathis Rocton: European Union's Horizon 2020 research and innovation COFUND programme (LogiCS@TUWien, grant agreement No 101034440).

1 Introduction

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 further down in this section. Note that only this step makes a difference between dominating set instances and hitting set instances. Once we have constructed the instance, we exhaustively apply a set of reduction rules described in Section 2. Some of these reduction rules split the instance into several smaller instances, in which case we solve these instances independently. After applying exhaustively the reduction rules, our algorithm uses several approaches, see Section 3. We use the library *htd* [1] 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 dynamic programming along the tree decomposition. 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. If none of these specialized approaches succeed, we finally create a MAX SAT instance equivalent to our current DIRECTED CONSTRAINED DOMINATION instance and use the *EvalMaxSAT* solver [2], with some improvements regarding the core computation.



© Alexander Dobler, Simon Dominik Fink, and Mathis Rocton;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Parameterized and Exact Computation (IPEC 2025).

Editors: Akanksha Agrawal and Erik Jan van Leeuwen; Article No. 35; pp. 35:1–35:5

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Directed Constrained Domination. We reduce both DOMINATING SET and HITTING SET to the following problem:

Problem DIRECTED CONSTRAINED DOMINATION

Input: A directed graph $G = (V, A)$, a *subsumed* set $S \subseteq V$ and a *dominated* set $D \subseteq V$.

Want: A set $DS(G) \subseteq V \setminus S$ of minimum size, such that $\forall v \in V \setminus D, \exists u \in DS(G), uv \in A$.

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, every edge is replaced by arcs in both directions, and no vertex is flagged as subsumed or dominated. Moreover, each vertex gets a self-loop.

For HITTING SET, create a dominated vertex for each element of the universe (i.e., the vertices of the hypergraph), and then a subsumed vertex for each set (i.e., each hyperedge). Add arcs from each dominated vertex to the subsumed vertices representing the sets it belongs to in the HITTING SET instance. Thus, any set will be hit if and only if the solution for DIRECTED CONSTRAINED DOMINATION contains a vertex belonging to this set. We have been made aware that a similar generalization of dominating set and some of our reduction rules have been presented in a recent IJCAI paper [10].

In the following, we assume any directed graph G to be implicitly accompanied by its labeling of subsumed and dominated vertices, and denote a solution to the corresponding DIRECTED CONSTRAINED DOMINATION problem as $DS(G)$. We use $[k]$ to denote $\{1, \dots, k\}$ for $k \in \mathbb{N}$. For a vertex v or set of vertices X , we define $N_{\text{out}}(v)$ and $N_{\text{out}}(X)$ as its out-neighborhood and $N_{\text{in}}(v)$ and $N_{\text{in}}(X)$ as its in-neighborhood (including self-loops).

2 Reduction Rules

Our algorithm first preprocesses the instance with a set of reduction rules to simplify the instance and make it smaller, thus making it easier to handle for the exact solvers called later on. The reduction rules we use are adapted to our DIRECTED CONSTRAINED DOMINATION problem and in parts generalized from the rules available for DOMINATING SET in the work of Van Rooij and Bodlaender [7] and Volkmann [9].

The following rule always has to be applied first to ensure the safeness of all other 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 classical reduction rules on vertices with extreme degrees and connected components to the directed setting. For the rules 3 and above, mind that due to Reduction Rule 1, the existence of an uv arc implies that u is not subsumed and v is not dominated.

► **Reduction Rule 2 (Out-degree 0).** *If there is a dominated vertex v with out-degree 0, remove v from the instance.*

► **Reduction Rule 3 (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 4 (Out-degree 1).** *Let v be a dominated vertex with a single outgoing arc vu . If $u \notin S$ or has at least in-degree 2, remove v . Otherwise, add v to the dominating set.*

► **Reduction Rule 5 (Conn. components).** *Connected components can be solved independently.*

► **Reduction Rule 6 (Subsumption).** *Let u, v be two vertices. If $N_{\text{out}}(u) \supseteq N_{\text{out}}(v)$, mark v as subsumed. Independently, if $N_{\text{in}}(u) \subseteq N_{\text{in}}(v)$, mark v as dominated.*

The efficient implementation of Reduction Rule 6 requires an adapted *partition refinement* algorithm [5] to compute the set-inclusion of neighborhoods. Essentially, we implemented this algorithm with directed arcs between the different partitions which are based on the in/out-neighborhoods of the vertices.

► **Reduction Rule 7 (Contraction).** *Let u, v be two vertices. If u is dominated but not subsumed, v is not dominated but subsumed, and there is an arc from u to v : mark u as not dominated, add to its in-neighborhood the in-neighborhood of v , and delete v .*

Proof. Observe that the corresponding MAX SAT instances (see Section 3) before and after applying the reduction rules are exactly the same. ◀

The following two rules, following principles identified in rules 6 and 7 of [7], require more care when translating to our setting. Our first special rule is a generalization of rule number 6, considering that we do not have control over the maximal size of out-neighborhoods, 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 6 between uses. Because of this, we need to make sure that some vertices do not have the same in-neighborhoods before reducing.

► **Reduction Rule 8 (Special 1 [7]).** *Let u be a vertex and Q be the out-neighbors of u having in-degree 2. Let $P = N_{in}(Q)$. If $\exists p \in P \setminus \{u\} : N_{out}(P) = N_{out}(P \setminus p)$, add u to the solution.*

► **Reduction Rule 9 (Special 2 [7]).** *Let v be a vertex with exactly 2 out-neighbors u_1, u_2 having in-degree 2 in G . 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 w with out-neighborhood the union of the out-neighborhoods of w_1 and w_2 , and subsume v, w_1 , and w_2 to obtain instance G' . If $w \in DS(G')$, we have $DS(G) = (DS(G') \setminus w) \cup \{w_1, w_2\}$ otherwise we have $DS(G) = DS(G') \cup \{v\}$.*

► **Reduction Rule 10 (Cut-vertices [9]).** *Let v be an (undirected) cut-vertex separating the subgraphs B_1, \dots, B_k in G , each including v . For $i \in [k]$, let B_i^- be the instance obtained from B_i by marking v dominated and let B_i^+ be B_i with $N_{out}[v]$ marked dominated. Then $DS(G)$ is the smallest of $\bigcup_i DS(B_i^+) \cup \{v\}$ and $\bigcup_{i \neq j} DS(B_i^-) \cup DS(B_j)$ for any $j \in [k]$.*

Note that, instead of applying this last reduction rule in one go, we instead iteratively apply it to individually remove leaf blocks of the block-cut-tree [4], and restrict this to blocks containing at most 25% of all vertices. This simplifies the implementation while avoiding multiple computations of a solution for large blocks, and also allows us to short-cut the computations in cases where we conclude that an optimal solution has already been found.

3 Exact Solvers and Other Improvements

Solving Low-Treewidth Instances. We use the *htd* library with the mindegree strategy to seek a nice tree-decomposition of small width [1]. If we obtain such a decomposition, we use a dynamic program to solve the instance in time $\mathcal{O}(3^{tw})$. To achieve this running time, our approach combines 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 0₀, whereas for introduce and forget nodes we consider the states 1, 0₁ and 0₀. 1 means the vertex is selected in the solution, 0₁ that it is not in the solution but neighbor to a vertex in the solution, 0₀ that it is neither in the solution nor has a neighbor in the solution. The additional 0_? 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 then translates 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.

Handling Vertex Cover Instances. We remark that some instances are encoding VERTEX COVER in the following sense: if all the vertices with incoming arcs (i.e., the vertices needing to be dominated) have in-degree exactly 2, the instance corresponds to an instance of VERTEX COVER where vertices with outgoing arcs are vertices, and vertices with ingoing arcs are edges between their two neighbors (a vertex can serve as an edge and a vertex simultaneously in this sense). For such instances with less than 1000 unsubsumed vertices, we run the exact solver *Peaty* for a maximum of 5 minutes [6].

EvalMaxSAT and Improvements. As final step we encode a remaining instance as SAT formula and use *EvalMaxSAT* to solve it [2]. The encoding is straight-forward, representing each vertex with a clause ensuring that at least one of its in-neighbors is selected. The selection of any vertex has a weight of -1 , meaning as few vertices as possible are selected.

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. Such a matching always provides an initial core for the Solver.

References

- 1 Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A free, open-source framework for (customized) tree decompositions and beyond. In *Proc. Integration of AI and OR Techniques in Constraint Programming (CPAIOR'2017)*, volume 10335 of *LNCS*, pages 376–386. Springer, 2017. <https://github.com/mabseher/htd>. doi:10.1007/978-3-319-59776-8_30.
- 2 Florent Avellaneda. Evalmaxsat 2023. *MaxSAT Evaluation 2023*, page 12, 2023. URL: <https://github.com/FlorentAvellaneda/EvalMaxSAT>.
- 3 Alexander Dobler, Simon Dominik Fink, and Mathis Rocton. The (not-so) Bad Dominating Set Maker. Software, Vienna Science and Technology Fund (WWTF) grant [10.47379/ICT19035], Vienna Science and Technology Fund (WWTF) grant [10.47379/ICT22029], European Union’s Horizon 2020 research and innovation COFUND programme (LogiCS@TUWien, grant agreement No 101034440), Austrian Science Fund (FWF) grant [10.55776/Y1329], sw-hId: sw-hId:1:dir:77b0720cd14b92e8d17e114ccf5c1ea89f79dfc9 (visited on 2025-12-05). URL: <https://github.com/Doblalex/pace2025>, doi:10.4230/artifacts.25245.
- 4 John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973. doi:10.1145/362248.362272.
- 5 Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987. doi:10.1137/0216062.
- 6 Patrick Prosser and James Trimble. Peaty: an exact solver for the vertex cover problem, 2019. URL: <https://github.com/jamestrimble/peaty>.
- 7 Johan M. M. van Rooij and Hans L. Bodlaender. Exact algorithms for dominating set. *Discret. Appl. Math.*, 159(17):2147–2164, 2011. doi:10.1016/J.DAM.2011.07.001.

- 8 Johan M. M. van Rooij, Hans L. Bodlaender, Erik Jan van Leeuwen, Peter Rossmanith, and Martin Vatshelle. Fast dynamic programming on graph decompositions. *CoRR*, abs/1806.01667, 2018. [arXiv:1806.01667](https://arxiv.org/abs/1806.01667).
- 9 Lutz Volkmann. A reduction principle concerning minimum dominating sets in graphs. *J. Comb. Math. Comb. Comp.*, 31:85–90, 1999.
- 10 Ziliang Xiong and Mingyu Xiao. Exactly solving minimum dominating set and its generalization. In *Proc. Conference on Artificial Intelligence, (IJCAI'2024)*, pages 7056–7064. [ijcai.org](https://www.ijcai.org), 2024. URL: <https://www.ijcai.org/proceedings/2024/780>.