



# PACE Solver Description: Shadoks Approach to Minimum Hitting Set and Dominating Set

Guilherme D. da Fonseca  

LIS, Aix-Marseille Université

Fabien Feschet  

LIMOS, Université Clermont Auvergne

Yan Gerard  

LIMOS, Université Clermont Auvergne

---

## Abstract

Description of the solvers used by the Shadoks team in the PACE 2025 challenge. The challenge considers solvers for the minimum dominating set and hitting set problems. For the heuristic challenge, we respectively won third and fourth place for hitting set and dominating set. For the exact challenge, we won fifth place on both problems.

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

**Keywords and phrases** Optimization, heuristic, hitting set, dominating set.

**Category** PACE Solver Description

**Supplementary Material** (*Source Code*): <https://github.com/gfonsecabr/shadoks-PACE2025>

**Funding** Work supported by the French ANR PRC grant ADDS (ANR-19-CE48-0005).

**Acknowledgements** We would like to thank the challenge organizers and other competitors for their time, feedback, and making this whole event possible.

## 1 Introduction

We describe the solvers used by the Shadoks team in the PACE 2025 challenge. The challenge considers solvers for the minimum dominating set and hitting set problems. Since it is easy to reduce minimum dominating set to minimum hitting set, we only consider the latter. We present both a heuristic and an exact algorithm for the problem, as our solution to both is linked in many aspects. For the heuristic challenge, we respectively won third and fourth place for hitting set and dominating set. For the exact challenge, we won fifth place on both problems.

Given a hypergraph  $H = (V, \mathcal{E})$  with vertices  $V$  and hyperedges  $\mathcal{E}$ , a *hitting set* is a subset  $S \subseteq V$  such that for all  $E \in \mathcal{E}$  we have  $S \cap E \neq \emptyset$ . A *minimum hitting set* is a hitting set with the smallest number of elements. To reduce a dominating set instance graph  $G = (V, E')$  to a hitting instance  $H = (V, \mathcal{E})$ , it suffices to set  $\mathcal{E}$  as the set of closed neighborhoods of the vertices in  $V$ .

In Section 2, we describe the different building blocks of our solvers. In Section 3, we show how the heuristic and exact solvers put these building blocks together. The source code of our solver is available at <https://github.com/gfonsecabr/shadoks-PACE2025>.

## 2 Tools and Techniques

**Reduction Rules:** There are three key reduction rules [6] that we apply.

1. If a hyperedge  $E$  is a superset of another hyperedge, then remove  $E$ .
2. If a vertex  $v$  hits a subset of another vertex, then remove  $v$ .

3. If a hyperedge has a single vertex  $v$ , then add  $v$  to the solution and remove all hyperedges that contain  $v$ .

For the large heuristic instances, we had to make the implementation of these reductions very efficient.

**Greedy Heuristic:** Greedy heuristics insert the vertex of maximum “value” in the solution at each step. The classic greedy heuristic sets the value of a vertex as the number of unhit hyperedges that it would hit. Since large hyperedges are likely to be hit anyway, we give a smaller weight to hitting large hyperedges. More precisely, given an estimate  $c$  of the cost of the solution (in our case, a previously found solution cost), the weight of a hyperedge  $E$  is  $w(E) = \left(1 - \frac{c}{|V|}\right)^{|E|-1}$ . The value of a vertex is the sum of the weights of the unhit hyperedges it hits.

**Local Search:** Local search replaces some vertices of the solution by a smaller (or equal sized) set of vertices, obtaining a new solution that is not larger than the previous one. We perform a large neighborhood local search by removing several vertices of the solution (that should not be too far from each other in the hypergraph), computing the hypergraph defined by the unhit hyperedges, reducing this hypergraph, and then solving it with a greedy heuristic or an exact algorithm.

**Clique and Independent Set:** We use the MCS [8] and MCQ [7] algorithms implemented in <https://github.com/darrenstrash/open-mcs>. Maximum cliques and independent sets appear in two places of our solver. The first one is inside EvalMaxSAT, where we replaced the original EvalMaxSAT maximum clique algorithm by MCQ for sparse graphs (density under 0.1) and MCS for dense graphs. There is a performance gain for the rare cases when EvalMaxSAT uses dense graphs, and a small performance loss otherwise, but the modern C++ implementation made it easier to code a time limit without risking memory leaks.

The second place where a maximum independent set solver is used is when the hitting set problem has all hyperedges of degree 2, which means we need to solve a minimum vertex cover problem. After performing the aforementioned reduction rules, there are 66 of the 200 public exact instances where the problems reduces to vertex cover. However, there are no instances where this happens among the 200 private exact instances.

**Exact MaxSAT:** It is easy to formulate minimum hitting set as a MaxSAT instance with one hard clause for each hyperedge and one variable (and soft clause) for each vertex. We used the 2024 version of EvalMaxSAT [2], which is based on the SAT solver Cadical [4]. We modified EvalMaxSAT in order to add a time limit to the computation. To make the task easier, we replaced the clique solver that EvalMaxSAT uses with the one mentioned above.

**Heuristic MaxSAT:** We also used the heuristic anytime MaxSAT solver tt-open-wbo-inc [5], especially the Nuwls solver in it [3]. This solver has not been modified and is called as a separate process with a timeout.

**Integer Programming:** Depending on the parameters used, EvalMaxSAT uses SCIP [1] for solving integer programming for around one minute before attempting to solve the problem directly. We noticed, however, that GLPK is significantly faster for small instances that can be solved in under half a second.

### 3 Our Solvers

Now that we have presented the main tools, we are ready to describe our solvers.

#### Heuristic Solver:

- 1) We apply the three reduction rules exhaustively.
- 2) We try to solve each connected component exactly with Integer Programming (GLPK) or MaxSAT (EvalMaxSAT) solvers, from small to large, aborting if taking too long. In fact, the greedy heuristic is used too, as it is optimal if the solution size is at most 2. If all components are solved, then we are done.
- 3) We reduce the problem to MaxSAT and run an anytime MaxSAT heuristic solver for around one to two minutes.
- 4) We then improve the solution using large neighborhood local search until we run out of time.

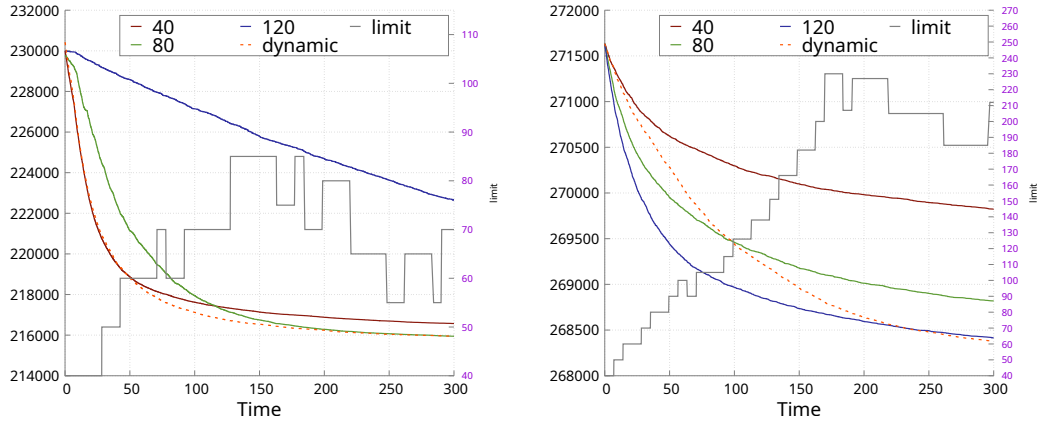
Step 4 deserves a more detailed explanation. In this step, we use a routine that removes vertices from our solution and then, replaces the removed part of the solution by a new set with a smaller or equal number of vertices. After removing several vertices from the solution, we obtain a set of unhit hyperedges. This set of unhit hyperedges (and all vertices in them) defines a smaller hypergraph instance. We apply the reduction rules to the smaller hypergraph and then try to solve it exactly with GLPK or EvalMaxSAT.

We tried different strategies to choose which vertices to remove, with no significant difference in the result, as long as the removed vertices are close to each other in the hypergraph. In our final version, we use the following strategy. We keep a queue of vertices and a vector of hyperedges. The queue is initialized with a random vertex of the solution and the vector is initially empty. At each step, we extract a vertex  $v$  from the queue and remove it from the solution if it is present there. We then add all hyperedges that contain  $v$  to the vector, making sure that each hyperedge is added only once. If the vertex queue is empty, we choose a random hyperedge from the vector and add all its vertices (that have not been added before) to the vertex queue.

A crucial part is the size of the smaller hypergraph, which is controlled by a *limit* parameter. We make sure that we remove at least  $limit/2$  vertices and get at least  $limit$  unhit hyperedges. Then, we use our main solver *i.e.* either EvalMaxSAT or GLPK on the smaller hypergraph. We regularly update the limit value as well as the main solver by independently testing GLPK and EvalMaxSAT on different values of *limit*. More precisely, we test three values  $limit$ ,  $limit + \delta$  and  $limit - \delta$  to determine whether *limit* should be increased, decreased, or kept, so that the ratio between the gain and the computation time is maximized. We also increase *limit* if no improvement is obtained, and decrease it if the solver times out without a solution. Figure 1 shows the evolution of the solution size using different values of *limit*.

#### Exact Solver:

- 1) We apply the three reduction rules exhaustively.
- 2) We try to solve each connected component exactly with Integer Programming (GLPK) or MaxSAT (EvalMaxSAT) solvers, from small to large, aborting if taking too long. If all components are solved, then we are done.
- 3a) If all hyperedges have degree 2 (and the number of vertices is at most 800), the problem is vertex cover and we solve it using the exact independent set solver MCS.



■ **Figure 1** Size of the solution over a 5-minute time interval with different values of *limit* for the private hitting set instances 002 and 072 respectively. The dashed line shows the version that dynamically changes *limit* and the corresponding value of *limit* is represented by the gray curve (scale in purple on the right).

- 3b) Otherwise, we find a good heuristic solution (as steps 3 and 4 of the heuristic solver) and give it as an initial solution to EvalMaxSAT. Step 3b is repeated twice with different parameters.

## References

- 1 Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009. doi:10.1007/s12532-008-0001-1.
- 2 Florent Avellaneda. A short description of the solver EvalMaxSAT. *MaxSAT Evaluation*, 8:364, 2020.
- 3 Yi Chu, Shaowei Cai, and Chuan Luo. Nuwls: Improving local search for (weighted) partial maxsat by new weighting techniques. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, volume 2023, pages 3915–3923, 2023. doi:10.1609/aaai.v37i4.25505.
- 4 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. CaDiCal, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020. *Artif. Intell.*, 301:103572, 2021. URL: <https://doi.org/10.1016/j.artint.2021.103572>, doi:10.1016/j.artint.2021.103572.
- 5 Alexander Nadel. Anytime algorithms for MaxSAT and beyond. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–1. IEEE, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6\_1.
- 6 Lei Shi and Xuan Cai. An exact fast algorithm for minimum hitting set. In *2010 Third International Joint Conference on Computational Science and Optimization*, volume 1, pages 64–67. IEEE, 2010.
- 7 Etsuji Tomita and Tomokazu Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *Discrete Mathematics and Theoretical Computer Science*, pages 278–289, 2003. doi:10.1007/3-540-45066-1\_22.
- 8 Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *International workshop on algorithms and computation*, pages 191–203, 2010. doi:10.1007/978-3-642-11440-3\_18.