Noname manuscript No.
(will be inserted by the editor)

The Revised Practitioner's Guide to MDP Model Checking Algorithms

Arnd Hartmanns · **Sebastian Junges** · **Tim Quatmann** · **Maximilian Weininger**

the date of receipt and acceptance should be inserted later

Abstract Model checking undiscounted reachability and expected-reward properties on Markov decision processes (MDPs) is key for the verification of systems that act under uncertainty. Popular algorithms are policy iteration and variants of value iteration; in tool competitions, most participants rely on the latter. These algorithms generally need worst-case exponential time. However, the problem can equally be formulated as a linear program, solvable in polynomial time. In this paper, we give a detailed overview of today's state-of-the-art algorithms for MDP model checking with a focus on performance and correctness. We highlight their fundamental differences, and describe various optimizations and implementation variants. We experimentally compare floating-point and exact-arithmetic implementations of all algorithms on three benchmark sets using two probabilistic model checkers. Our results show that (optimistic) value iteration is a sensible default, but other algorithms are preferable in specific settings. This paper thereby provides a guide for MDP verification practitioners—tool builders and users alike.

Keywords Quantitative model checking · Markov decision process · Linear programming · Value iteration · Policy iteration

This research was funded by the European Union's Horizon 2020 research and innovation programme under Marie Skłodowska-Curie grant agreements 101008233 (MISSION) and 101034413 (IST-BRIDGE), by the Interreg North Sea project STORM_SAFE, by a KI-Starter grant form the Ministerium für Kultur und Wissenschaft NRW, and by NWO VENI grant no. 639.021.754. Simulations were performed with computing resources granted by RWTH Aachen University under project rwth1632.

Data availability statement. The datasets generated and analysed in this study and code to regenerate them will be publicly archived at Zenodo.

A. Hartmanns: University of Twente, Enschede, The Netherlands
E-mail: a.hartmanns@utwente.nl
S. Junges: Radboud University, Nijmegen, The Netherlands
E-mail: sebastian.junges@ru.nl
T. Quatmann: RWTH Aachen University, Aachen, Germany
E-mail: tim.quatmann@cs.rwth-aachen.de
M. Weininger: Technical University of Munich, Munich, Germany and Institute of Science and Technology Austria, Klosterneuburg, Austria
E-mail: mweining@ist.ac.at

1 Introduction

The verification of Markov decision processes (MDPs) is crucial for the design and evaluation of cyber-physical systems with sensor noise, biological and chemical processes, network protocols, and many other complex systems. MDPs are the standard model for sequential decision making under uncertainty and thus at the heart of reinforcement learning. Many dependability evaluation and safety assurance approaches rely in some form on the verification of MDPs with respect to temporal logic properties. Probabilistic model checking [7,10] provides powerful tools to support this task.

The essential MDP model checking queries are for the *worst-case probability that something bad happens* (reachability) and the *expected resource consumption until task completion* (expected rewards). These are *indefinite (undiscounted) horizon* queries: They ask about the probability or expectation of a random variable up until an event—which forms the horizon—but are themselves unbounded. Many more complex properties internally reduce to solving either reachability or expected rewards. For example, if the description of *something bad* is in linear temporal logic (LTL), then a product construction with a suitable automaton reduces the LTL query to reachability [11]. This paper sets out to determine the practically best algorithms to solve indefinite

horizon reachability probabilities and expected rewards; our methodology is an empirical evaluation.

MDP analysis is well studied in many fields and has lead to three main types of algorithms: *value iteration* (VI), *policy iteration* (PI), and *linear programming* (LP) [76]. While indefinite horizon queries are natural in a verification context, they differ from the standard problem of e.g. operations research, planning, and reinforcement learning. In those fields, the primary concern is to *compute a policy* that (often approximately) optimizes the *discounted* expected reward over an infinite horizon where rewards accumulated in the future are weighted by a discount factor < 1 that exponentially prefers values accumulated earlier.

The lack of discounting in verification has vast implications. The *Bellman operation*, essentially describing a one-step backward update on expected rewards, is a contraction with discounting, but not a contraction without. This leads to significantly more complex termination criteria for VI-based verification approaches [49]. Indeed, VI runs in polynomial time for every fixed discount factor [68], and similar results are known for PI as well as LP solving with the simplex algorithm [81]. In contrast, VI [15] and PI [32] are known to have exponential worst-case behaviour in the undiscounted case.

So, *what is the best algorithm for model checking MDPs?* A (weakly) polynomial-time algorithm exists using an LP formulation and barrier methods for its solution [19]. LP-based approaches (and their extension to MILPs) are also prominent for multi-objective model checking [35], in counterexample generation [37], and for the analysis of parametric Markov chains [26]. However, folklore tells us that iterative methods, in particular VI, are better for solving MDPs. Indeed, variations of VI are the default choice of all model checkers participating in the QComp competition [22]. This uniformity may be misleading. Indeed, for some stochastic game algorithms, using LP to solve the underlying MDPs may be preferential [5, Appendix E.4]. An application in runtime assurance preferred PI for numerical stability [61, Sect. 6]. A toy example from [49] is a famous challenge for VI-based methods. Despite the prominence of LP, the ease of encoding MDPs, and the availability of powerful off-the-shelf LP solvers, many tools did (until very recently) not include MDP model checking via LP solvers.

With this paper, we reconsider the PI and LP algorithms to investigate whether probabilistic model checking focused on the wrong family of algorithms. Indeed, a similar comparison of algorithms on the more general model of stochastic games [64] revealed that, depending on the considered case study, VI or SI can be preferable.[1]

We report the results of an extensive empirical study with two independent implementations in the model checkers Storm [57] and mcsta [52]. We also emphasize the question of precision and soundness. Numerical algorithms, in particular ones that converge *in the limit*, are prone to delivering wrong results. For VI, the recognition of this problem has led to a series of improvements over the last decade [13,20,31,49,54,75,77]. We show that PI faces a similar problem. When using floating-point arithmetic, additional issues may arise [51,80]. Our use of various LP solvers also exhibits concerning results for a variety of benchmarks. We therefore include results for *exact* computation using rational arithmetic as well.

Limitations of this study. A thorough experimental study of algorithms requires a carefully scoped evaluation. We work with flat representations of MDPs that fit completely into memory (i.e. we ignore the state space exploration process and symbolic methods). We selected algorithms that are tailored to converge to *the* optimal value. We also exclude approaches that incrementally build and solve (partial or abstract) MDPs using simulation or model checking results to guide exploration: they are an orthogonal improvement and would equally profit from faster algorithms to solve the partial MDPs. Moreover, this study is on algorithms, not on their implementations. To reduce the impact of potential implementation flaws, we use two independent tools where possible.

We consider three classes of algorithms, each with many hyper-parameters, for example using different variant of the algorithm, different solvers and different optimization. Overall, this results in exponentially many tool-configurations, and it is infeasible to run all of these on all benchmarks. Thus, we select a representative set of benchmarks (called *practitioner-set-2024*, see below), and do not analyze all possible interactions of hyper-parameters. Instead, we investigate each hyper-parameter separately, choosing reasonable values for the others, usually based on other experiments.

Contributions. This paper contributes a thorough overview on how to model-check indefinite horizon properties on MDPs, making MDP model checking more accessible, but also pushing the state-of-the-art by clarifying open questions. We provide new insights and review folklore statements. Particular highlights are:

- a discussion of the practical guarantees of different solution methods. We provide a new simple but challenging MDP family that leads to wrong results on all floating-point LP solvers (Section 2.3), a discussion of the practical impact of precision parameters in state-

[1] The implementations used in that comparison were often prototypical, not mature tools like we consider in this paper. Thus, the difference in performance could also be due to engineering, not due to fundamental properties of the algorithms.

of-the-art LP solvers (Section 3.2), a negative result regarding the soundness of PI with epsilon-precise policy evaluators (Section 4), and performance comparisons of floating-point precise and exact methods (throughout Section 6).
– a description of the known classes of solution algorithms for MDPs (Section 2.2), together with various prominent optimizations and preprocessing techniques (Section 2.4) and algorithm-specific improvements (Section 3 for LP and Section 4 for PI). Further, we thoroughly evaluate of their practical impact of these hyper-parameters (Section 6.1). This reveals that many things which are considered optimizations are not universally beneficial in practice.
– an empirical evaluation of many factors which can affect the outcome of a comparison of algorithms, hyper-parameters and tools. This includes using two independent code bases for probabilistic model checkers (Storm and mcsta) and 10 different LP solvers. Further, we analyze the influence of the selected benchmark set (Section 6.2), the state ordering used inside model checkers (Section 6.3) and the influence of hardware (Section 6.4).

Overall, we find that there is no simple answer to the question "What is the best algorithm for model checking MDPs?" Depending on the considered benchmark instance, different algorithms and optimizations perform best. We summarize the insights of our empirical evaluation in Section 6.5, giving recommendations for how to choose an algorithm when solving MDPs.

Differences to conference version. This paper is the journal version of [53] and essentially constitutes a complete rewrite of that paper and renewal of its experimental evaluation. We have made the paper more accessible by including more explanations, examples and references. Moreover, we widened the scope of the experimental evaluation, including more optimizations, as well as factors such as the tolerance parameters of LP solvers, the state ordering, or hardware. We also updated all LP solvers to their newest versions.

Finally, we establish a new set of benchmark instances called the *practitioner-set-2024*. The idea of this set is that, on the one hand, it is small enough to be run in reasonable time on any machine, thus allowing others to quickly compare further algorithm variants. On the other hand, it is large enough to be structurally diverse and include many "interesting" cases, e.g. those where some algorithms are incorrect, where differences between the algorithms become apparent, or where algorithms have to deal with additional complications such as maximal end components. We in particular note that the experimental evaluation of the conference version did not include models with nontrivial end components.

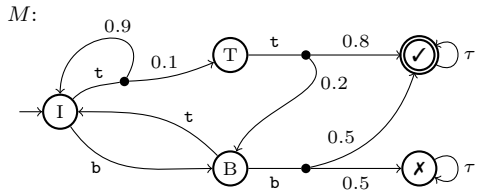


Fig. 1: An example MDP

We use the *practitioner-set-2024* to validate the conclusions drawn in the conference version, and we hope that it will simplify the job of developers of algorithms and tools in the future. We included the current year "2024" in the name of our new benchmark set: We expect it to be modified and improved in the future, when new benchmarks are constructed and new structures are discovered to be relevant.

2 Background

We recall MDPs with reachability and reward objectives, describe solution algorithms and their guarantees, and address commonly used optimizations.

2.1 Markov Decision Processes

A probability distribution over a set X is a function $\mathsf{d}\colon X \to [0,1]$ such that for all $x \in X$ we have $0 \geq \mathsf{d}(x) \geq 1$ and $\sum_{x \in X} \mathsf{d}(x) = 1$. We denote the set of all probability distributions over X by D_X. A Markov decision process is a probabilistic transition system model that includes nondeterministic choices between actions followed by probabilistic choices of successor states:

Definition 2.1 A *Markov decision process* (MDP) [16, 76] is a tuple $\mathcal{M} = (\mathsf{S}, \mathsf{A}, \delta)$ consisting of finite sets of states S and actions A, and a partially defined transition function $\delta\colon \mathsf{S} \times \mathsf{A} \rightharpoonup \mathsf{D}_\mathsf{S}$.

The transition function δ maps enabled state-action pairs to distributions over successor states. We overload A to also denote the function that assigns the set of enabled actions to every state, namely $\mathsf{A}(s) := \{\, a \mid (s,a) \in \mathit{domain}(\delta) \,\}$ for all $s \in \mathsf{S}$, and we require this set to be non-empty for all states (i.e. we require deadlock freedom). A Markov chain (MC) is an MDP with $|\mathsf{A}(s)| = 1$ for all s.

Example 2.1 We show an example MDP $M = (\mathsf{S}, \mathsf{A}, \delta)$ in Figure 1. Its has five states, $S = \{\,\mathrm{I}, \mathrm{T}, \mathrm{B}, ✓, ✗\,\}$, represented by the circles. Its set of actions is $\mathsf{A} = \{\,\mathtt{t}, \mathtt{b}, \tau\,\}$, and each line leaving a circle represents an enabled action of the corresponding state. We thus have $\mathsf{A}(I) =$

$\mathsf{A}(B) = \{\,\mathtt{t}, \mathtt{b}\,\}$, $\mathsf{A}(T) = \{\,\mathtt{t}\,\}$, and $\mathsf{A}(✓) = \mathsf{A}(✗) = \{\,\tau\,\}$. The distributions that δ maps the enabled state-action pairs to are indicated after the solid • dots; where the dot is omitted, the distribution is a Dirac distribution, i.e. it assigns probability 1 to a single successor state. We have, for example, $\delta(\mathrm{B}, \mathtt{t})(I) = 1$, $\delta(\mathrm{T}, \mathtt{t})(✓) = 0.8$, and $\delta(\mathrm{T}, \mathtt{t})(\mathrm{B}) = 0.2$. The distributions assign probability 0 to all successor states to which no arrow is drawn, e.g. $\delta(\mathrm{B}, \mathtt{t})(T) = 0$. The transition function is undefined for non-enabled actions; e.g. $\delta(\mathrm{I}, \tau)$ and $\delta(\mathrm{T}, \mathtt{b})$ are undefined.

M models a scenario where the player choosing the actions can pick between two routes towards ✓: the top route via action `t` and the bottom route via action `b`. The top route has a good chance of bringing the player to ✓, but this may take a while—in particular due to the high probability of the `t`-transitions out of state I to loop back to I. The bottom route, on the other hand, is fast, but has a high risk of leading into the dead-end ✗ state instead of ✓. △

The *semantics* of an MDP is defined in the usual way by means of policies and the unique probability measure over paths which they induce. We briefly recall the most important concepts, summarizing [11, Chapter 10]: A (memoryless deterministic) policy—a.k.a. strategy, adversary or scheduler—is a function $\pi\colon \mathsf{S} \to \mathsf{A}$ that, intuitively, given the current state s prescribes which action $a \in \mathsf{A}(s)$ to play [11, Definitions 10.91 and 10.96]. Applying a policy π to an MDP induces an MC $\mathcal{M}^\pi$ [11, Definition 10.92].

Example 2.2 Considering again the MDP in Figure 1, an example policy is $\pi(s) = \begin{cases} \mathtt{t} & \text{if } s \in \{\mathrm{I}, \mathrm{T}, \mathrm{B}\} \\ \tau & \text{otherwise} \end{cases}$, which picks the only available action τ in states ✓ and ✗ and `t` in all other states. Applying this strategy to the MDP yields an MC, as all non-determinism is resolved. Intuitively, we get the MC by removing the arrows labelled with `b` from states I and B.[2] △

A path in this MC is an infinite sequence $\rho = s_1 s_2 \ldots$ with $\delta(s_i, \pi(s_i))(s_{i+1}) > 0$. Paths denotes the set of all paths and $\mathbb{P}^\pi_s$ denotes the unique probability measure of $\mathcal{M}^\pi$ over infinite paths starting in the state s, see [11, Chapter 10.1], in particular the Excursus on Probability Spaces and the following discussion.

Example 2.3 An example of a path through the induced MC from Example 2.2 is $\rho = \mathrm{I}\ \mathrm{I}\ \mathrm{T}\ ✓^\omega$. Note that it is an infinite path that cycles forever in state ✓. This path has the probability $\mathbb{P}^\pi_\mathrm{I}[\{\rho\}] = 0.9 \cdot 0.1 \cdot 0.8 = 0.072$. △

A *reachability objective* $\mathrm{P}_{\mathsf{opt}}(\mathsf{T})$ with set of target states $\mathsf{T} \subseteq \mathsf{S}$ and $\mathsf{opt} \in \{\max, \min\}$ induces a random variable $X\colon \mathsf{Paths} \to [0,1]$ over paths by assigning 1 to all paths that eventually reach the target and 0 to all others [11, Chapter 10.6.1]. $\mathrm{E}_{\mathsf{opt}}(\mathsf{rew})$ denotes an *expected reward objective*, where $\mathsf{rew}\colon \mathsf{S} \to \mathbb{Q}_{\geq 0}$ assigns a reward to each state. $\mathsf{rew}(\rho) := \sum_{i=1}^{\infty} \mathsf{rew}(s_i)$ is the accumulated reward of a path $\rho = s_1 s_2 \ldots$. This yields a random variable $X\colon \mathsf{Paths} \to \mathbb{Q}_{\geq 0} \cup \{\infty\}$ that maps paths to their reward [14]. For a given objective and its random variable X, the *value of a state* $s \in \mathsf{S}$ is the expectation of X under the probability measure $\mathbb{P}^\pi_s$ of the the MC induced by an optimal policy π from the set of all policies Π, formally $\mathsf{V}(s) := \mathsf{opt}_{\pi \in \Pi} \mathbb{E}^\pi_s[X]$.

Example 2.4 In Fig. 1, state ✓ is marked as the target state of a reachability objective, and hence all paths reaching it are assigned value 1. The value of the reachability objective $\mathrm{P}_{\max}(\{✓\})$ is the highest possible probability to reach ✓ under an optimal policy π^*. In fact, π from Example 2.2 is an optimal policy, and it achieves a value of 1. Intuitively, we almost surely reach ✓, because even if action `t` in state T is unlucky and proceeds to B, the path returns to I, eventually reaches T again and has another chance of reaching the target.

We mention several technicalities: Firstly, the definition of the value of an MDP typically uses sup and inf over the policies, rather than max and min. However, for the objectives we consider, memoryless deterministic policies suffice, see [11, Lemma 10.102 and 10.113] for reachability and [18, Proposition 2] for expected reward. Thus, as there are only finitely many of these policies, we can equivalently use max and min. Secondly, we only allow rew to assign non-negative rewards. If state-rewards are both positive and negative, the reward of a path is not well-defined [76, Chapter 7.1.1]. Finally, we assume that the expected reward is finite (sometimes enforced by assuming that all policies are *proper* [18]); this is without loss of generality, as states with infinite reward can be detected by graph algorithms and removed as part of preprocessing, see [24, Section 4.3].

2.2 Solution Algorithms

We briefly recall the classes of solution algorithms used for computing the value of MDPs and complement them with references to extensive descriptions and examples.

Value iteration (VI) works by iteratively updating vectors of estimates, i.e. functions $x\colon \mathsf{S} \to \mathbb{Q}$ that assign to

[2] Note that [11, Definition 10.92] constructs a countably infinite MC as it deals with general policies. In our setting, we can restrict to memoryless deterministic policies, and thus the resulting MC has exactly the same set of states as the original MDP.

every state an estimate of the value. Given such a function, the algorithm applies so-called *Bellman updates*, which intuitively execute one step of the MDP. Formally, for reachability and expected reward objectives, the Bellman update is defined as

$$x_{i+1}(s) = \mathsf{rew}(s) + \mathsf{opt}_{a \in \mathsf{A}(s)} \sum_{s' \in \mathsf{S}} \delta(s,a)(s') \cdot x_i(s'),$$

where opt is the optimization direction and in the case of reachability objectives, the state reward $\mathsf{rew}(s)$ is 0 and thus can be omitted.

VI starts from an estimate-vector that is a safe under- or over-approximation of the value. For example, for a maximizing reachability objective we can use the under-approximation that assigns the smallest probability 0 to all non-target states and 1 to targets. Then, repeatedly applying Bellman updates converges to the value in the limit. We refer to Algorithm 5 and Example 6 in [34] for very concrete pseudocode and an illustrative example

Many variants of VI have been developed, aiming to improve its practical performance. We do not discuss them here, but refer to the respective papers for Gauss-Seidel VI [34], interval iteration [49], partial-exploration based VI [20], sound VI [77], and optimistic VI [54]. Further, [23] surveys how VI can be applied in many different settings, going beyond MDPs.

Linear programming (LP) [11, Chapter 10.6.1] encodes the transition structure of the MDP and the objective as a linear optimization problem. For every state, the LP has a variable representing an estimate of its value. Every state-action pair is encoded as a constraint on these variables, as are the target set or rewards. The unique optimum of the LP is attained if and only if for every state its corresponding variable is set to the value of the state. We provide an in-depth discussion of theoretical and practical aspects of LP in Section 3.

Policy iteration (PI) computes a sequence of policies converging to the optimal one. For the description, consider an objective with optimization direction opt and random variable over paths X. PI starts from an arbitrary policy π_0. Then we repeat the following steps:

- For every state s, compute the value in the MC induced by π_i, i.e. $p_s := \mathbb{E}^{\pi}_s[X]$.
- For every state s, construct a new policy π_{i+1} by choosing an action that is locally optimal. Formally, set $\pi_{i+1}(s) := \arg \mathsf{opt}_{a \in \mathsf{A}(s)} \sum_{s' \in \mathsf{S}} \delta(s,a)(s') \cdot p_{s'}$.

We repeat these steps until $\pi_i = \pi_{i+1}$. The algorithm only considers memoryless deterministic policies, of which there are only finitely many (a number exponential in the number of states). Thus, since every iteration strictly improves the policy, PI terminates in finite time, having computed an optimal policy as well as its value (namely the value in its induced MC).

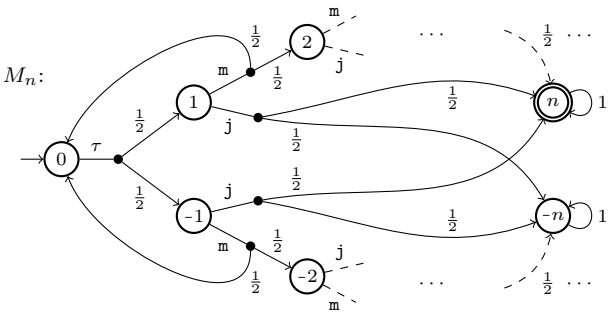


Fig. 2: A hard MDP for all algorithms

We refer to Algorithm 7 and Example 7 in [34] for very concrete pseudocode and an illustrative example, and to [4] for a recent generic policy iteration framework surveying and including lots of variants of the algorithm. We further discuss practical consideration regarding PI in Section 4.

2.3 Guarantees

Given the stakes in many application domains, we require guarantees about the relation between an algorithm's result $\bar{v}$ and the true value v. First, implementations are subject to floating-point errors and imprecision [80] unless they use exact (rational) arithmetic or safe rounding [51]. This can result in arbitrary differences between $\bar{v}$ and v.

Second are the algorithm's inherent properties: VI is an approximating algorithm that converges to the true value only in the limit. In theory, it is possible to obtain the exact result by rounding after exponentially many iterations [23]; in practice, this results in excessive runtime. Instead, for years, implementations used a naive stopping criterion that could return arbitrarily wrong results [48]. This problem's discovery sparked the development of sound variants of VI [3,13,31,49,54,75,77]. A sound VI algorithm guarantees ε-precise results, i.e. $|v - \bar{v}| \leq \varepsilon$ or $|v - \bar{v}| \leq v \cdot \varepsilon$ (absolute and relative difference ε, respectively).

For LP and PI, the guarantees have not yet been thoroughly investigated. Theoretically, both are exact, but implementations are often not. We discuss the problems in detail in Sections 3 and 4 and here exemplify the lack of guarantees using Figure 2.

Example 2.5 Figure 2 shows a more complex MDP M_n, which is an extension of the MC introduced in Figure 2 of [48]. It is parameterized by $n \in \mathbb{N}$, $n \geq 2$, which can be used to scale the size of the model. M_n contains two chains of states of length n, one positive and one

Table 1: Correct results

alg.	solver	$n \leq$
PI	–	20
LP	COPT	18
	CPLEX	18
	Glop	25
	GLPK	24
	Gurobi	18
	HiGHS	22
	lp_solve	28
	Mosek	22
	SoPlex	34

negative, each leading an absorbing state. Inside the chains, action m **m**oves one step forward in the chain, but has a chance of $\frac{1}{2}$ to return to the initial state. Using action j immediately **j**umps to the end of the chain, with a 50-50 chance of ending in the positive or the negative state. The addition of this jump action is the difference to the original model of [48, Figure 2].

The original MC (using only the move action) is hard for VI because the convergence rate is extremely slow. For example, after n iterations the estimate for state 0 is increased for the first time, namely to $(\frac{1}{2})^n$.

By adding the **j**ump action to every state, we layered a "deceptive" decision problem on top of the slow convergence of the original MC: Initially, using j everywhere looks like the best policy for $\mathrm{P}_{\max}$. As updated values slowly propagate, state-by-state, `m` becomes the optimal choice in all states except $-(n-1)$. Using this optimal policy, when reaching the end of the positive chain, we either win (reaching n) or return to the start; and when reaching the end of the negative chain, we have a 50-50 chance of winning or loosing. Thus, the resulting expected probability to reach n from state 0 is $\frac{2}{3}$. Dually, when minimizing the value, we can choose m in all states except $(n-1)$ to obtain $\frac{1}{3}$. △

We evaluated whether the implementations of the algorithms, in particular of PI and LP which are exact in theory, provide the necessary guarantees in practice. For this, we ran the algorithms on the MDP just described, using default settings for all tools and solvers. Overall, for large enough n, all non-exact implementations of algorithms return incorrect results. Storm's exact-arithmetic engine produces correct results, though at increased runtime.

For $n = 20$, naive VI with Storm and mcsta deliver the incorrect results $(\mathrm{P}_{\min}(\{n\}), \mathrm{P}_{\max}(\{n\})) = (0.247, 0.500)$ (whereas the optimal values over all policies are $(\frac{1}{3}, \frac{2}{3})$, see Example 2.5). Sound VI algorithms return ε-correct results, though at excessive runtime for larger n. In theory, for large enough n, floating-point sound VI cannot converge to the correct result anymore since the estimates increase by less than floating-point precision.

For Storm's PI implementations and for Storm's and mcsta's LP implementations using various LP solvers, we show in Table 1 the largest n for which each method or solver returns a ± 0.01-correct result. For larger n, PI and all LP solvers claim $\approx (\frac{1}{2}, \frac{1}{2})$ as the correct solution; Glop and GLPK are exceptions, as they fail only for the $\mathrm{P}_{\max}(\{n\})$ at the given n, and return a wrong result for $\mathrm{P}_{\min}(\{n\})$ at $n \geq 29$ and 52, respectively.

2.4 Optimizations

Solving MDPs may benefit from several optimizations. We provide an overview of those we are aware of and that have relevance in current tools. We roughly order the optimisations in terms of commonality and necessity: The first four optimisations, from qualitative precomputations to MEC collapsing, are commonly implemented in probabilistic model checking (PMC) tools today or relatively easy to implement, and in part necessary for the correctness or termination of certain algorithms. The later optimisations in our list are currently supported by fewer tools and may in part significantly alter the structure of the MDP structure. Our experimental evaluation in Section 6.1 includes all the first four optimisations as well as essential states and bisimulation reduction. Section 6.1.4 summarises our observations.

Qualitative precomputations [34, Section 4.1] can be used for finding states with value 0 or (only for reachability objectives) 1. These precomputations are often much faster than the numerical computations for quantitative analysis since they rely solely on the topology of the underlying graph. Thus, tools can apply them first and only run the numerical algorithms on the remaining states with non-trivial values. For example, the MDP in Fig. 1 can be solved solely by qualitative methods.

Warm starts, e.g. [40,64], may adequately initialize an algorithm, i.e., provide prior knowledge so that the computation has a good starting point. In our experiments, we evaluate warm starts by first running VI for a limited number of iterations and using the resulting estimate to guess bounds on the variables in LP or a good initial policy for PI; see Sections 3 and 4 for more details.

Optimistic VI can also be seen as interval iteration with a VI warm start: it first performs VI to guess an upper estimate vector that is closer to optimal. VI itself could also benefit from prior knowledge on the value vector, but it is unclear how to efficiently obtain such knowledge seeing as VI is currently the go-to method to obtain prior knowledge.

Topological methods [27,6] do not consider the whole MDP at once. Instead, they first compute a topological ordering of the strongly connected components (SCCs). A set $\mathsf{S}' \subseteq \mathsf{S}$ is a connected component if for all $s, s' \in \mathsf{S}'$, s can be reached from s'. We call S' strongly connected component if it is inclusion-maximal. The values of states in an SCC depend only on the values of states in SCCs that come later in the topological ordering. In particular, *bottom* SCCs at the end of the ordering have a trivial value of either 0 or 1. Using this, topological methods solve individual SCCs, building on the already computed values of SCCs later in the topological ordering. Intuitively, this decreases the runtime, since the values on the exits from an SCC have already been computed precisely. Note that we can use any algorithm for solving the individual SCCs.

Collapsing of maximal end components (MECs) [49,20] transforms the MDP into one with equivalent values but simpler structure. A set of states $\mathsf{S}' \subseteq \mathsf{S}$ is an end component if there exists a policy π such that, in the MC induced by π, from all states $s \in \mathsf{S}'$ the probability to reach every other state $t_1 \in \mathsf{S}'$ is positive and the probability to reach some state $t_2 \in \mathsf{S} \setminus \mathsf{S}'$ is zero. We call S' a maximal end component if there is no superset of S' that is an end component. After replacing every MEC by a single state, the MDP is contracting, i.e. we almost surely reach a target state or a state with value zero. VI algorithms rely on this property for convergence [49, 54,77]. For PI and LP, simplifying the graph structure before applying the solution method can speed up the computation.

Cores [63] are subsets of states in which all paths provably remain with high probability $1-\varepsilon$. Thus, a core can be used to prove properties with ε-precision, avoiding computations on the rest of the state space. A core can also be computed on-the-fly, leading to partial-exploration based algorithms [20,71]. We do not consider such algorithms in our experiments, as they are an orthogonal improvement and would equally profit from faster algorithms to solve the partial MDPs. We refer to [71,72] for more details and empirical comparisons, which in particular show that, depending on the model structure, using partial exploration can improve or worsen the performance.

The essential states reduction [28] computes a *domination relation* over the MDP's states as a least fixed point of the condition that state s is dominated by state t if, for all $a \in \mathsf{A}(s)$ and all s' such that $\delta(s,a)(s') > 0$, state s' is also dominated by state t. An essential state is a state not dominated by any other state. Intuitively, non-essential states reach an essential state with minimum probability 1 in a finite number of steps. The MDP can then be reduced to an equivalent one consisting only of the essential states, where equivalence means the preservation of reachability and, with a straightforward extension, expected reward properties. The domination relation is easy to compute, and despite the simplicity of the underlying condition, many of the models we consider contain far fewer essential than non-essential states.

The never-worse relation (NWR) [78] can similarly be used to merge states that behave equivalently. A state s is never worse than a state s' if the value of s is at least that of s', independent of the actual transition probabilities $\delta(s,a)(s')$. As such, the analysis is based only on the graph structure of the MDP. States that are mutually never worse form an equivalence class, and thus can be merged without changing the value of the MDP. Further, the NWR allows identification and removal of some suboptimal actions. Using qualitative precomputations, collapsing MECs, and reducing to essential states are all special cases of merging states that form an equivalence class with respect to the NWR. However, computing the whole NWR is **coNP**-complete. Thus, the simpler special cases remain practically relevant; we are not aware of PMC tools implementing the full NWR and therefore do not consider it in our experiments.

Game-based abstraction [62] reduces the size of the state space by computing an abstraction of the MDP. This abstraction takes the form of a turn-based stochastic game. As we focus on methods for solving MDPs, not stochastic games, we do not consider this optimization in our experimental evaluation. Still, using the abstraction-refinement loop provided in [62] and then employing efficient algorithms for stochastic games, cf. [64], is an interesting direction of future work.

Partial order reduction [25, Chapter 6] is a well-established technique in verification to reduce the size of a model. It exploits the observation that models in higher-level modelling languages are typically defined as a composition of concurrently executing modules, and that the order of independently-performed transitions often does not matter. Then, keeping one representative for all possible orderings suffices, and many transitions—as well as the states that only they lead to—can be removed. Partial order reduction has been extended to probabilistic systems [8,9,29], where more restrictive conditions apply than in the traditional non-probabilistic setting. Prototype implementations showed significant reduction potential for certain models [41], but no widely-used PMC tool today offers partial-order reduction. We thus do not include this technique in our experiments.

Bisimulation minimization [67] merges states with equivalent behavior, potentially allowing for a significant state

space reduction. Intuitively, two states $s, t \in S$ are bisimilar, written $s \sim t$, if they can mimic each others transitioning behaviour. Formally, assuming a reachability objective $\mathrm{P}_{\mathsf{opt}}(\mathsf{T})$, bisimilarity $\sim$ is defined as the coarsest equivalence relation $R \subseteq S \times S$ such that $(s,t) \in R$ iff $s \in \mathsf{T} \iff t \in \mathsf{T}$ and for all actions $a \in \mathsf{A}(s)$ there is an action $b \in \mathsf{A}(t)$ such that for all equivalence classes $C \in S/R$: $\sum_{s' \in C} \delta(s,a)(s') = \sum_{t' \in C} \delta(t,b)(t')$. For expected reward objectives, bisimilar states additionally need to yield the same reward value. The quotient MDP $\mathcal{M}/_\sim$ merges bisimilar states, i.e., its state-space coincides with the equivalence classes of $\sim$. Since this merging preserves objective values, one can solve the (potentially much smaller) quotient MDP instead of the original model. The relation $\sim$ can be computed using a partition refinement algorithm. Our experiments in Section 6.1.4 consider a symbolic implementation using a MTBDD-based representation of the MDP $\mathcal{M}$ [56].

3 Solving MDPs with Linear Programs

This section considers the LP-based approach to solving the optimal policy problem in MDPs. To the best of our knowledge, this is the only known polynomial-time approach. We discuss various configurations. These configuration are a combination of the LP formulation and the choice of LP solver implementations and their hyperparameters.

3.1 How to Encode MDPs as LPs?

For objective $\mathrm{P}_{\max}(\mathsf{T})$ we formulate the following LP over variables x_s, $s \in \mathsf{S} \setminus \mathsf{T}$:

$$\begin{aligned} \text{minimize} \quad & \sum_{s \in \mathsf{S}} x_s \\ \text{such that} \quad & lb(s) \le x_s \le ub(s) && \text{(for all } s \in \mathsf{S}\text{)} \\ & x_s = 1 && \text{(for all } s \in \mathsf{T}\text{)} \\ & x_s \ge \sum_{s' \in \mathsf{S}} \delta(s,a)(s') \cdot x_{s'} \\ & && \text{(for all } s \in \mathsf{S} \setminus \mathsf{T}, a \in \mathsf{A}(s).\text{)} \end{aligned}$$

We assume bounds $lb(s) = 0$ and $ub(s) = 1$ for $s \in \mathsf{S} \setminus \mathsf{T}$. The unique solution $\eta \colon \{ x_s \mid s \in \mathsf{S} \setminus \mathsf{T} \} \to [0,1]$ to this LP coincides with the desired objective values $\eta(x_s) = V(s)$ [11, Theorem 10.105]. Objectives $\mathrm{P}_{\min}(\mathsf{T})$ and $\mathrm{E}_{\mathsf{opt}}(\mathsf{rew})$ have similar encodings: minimizing policies require maximization in the LP and flipping the constraint relation. Rewards can be added as an additive factor on the right-hand side. We exemplify these changes by also providing the LP for the objective $\mathrm{E}_{\min}(\mathsf{rew})$ (where $ub(s)$ can be infinite or approximated using [13, Section 3.3]).

$$\begin{aligned} \text{maximize} \quad & \sum_{s \in \mathsf{S}} x_s \\ \text{such that} \quad & lb(s) \le x_s \le ub(s) && \text{(for all } s \in \mathsf{S}\text{)} \\ & x_s = 0 && \text{(for all } s \in \mathsf{T}\text{)} \\ & x_s \le \mathsf{rew}(s) + \sum_{s' \in \mathsf{S}} \delta(s,a)(s') \cdot x_{s'} \\ & && \text{(for all } s \in \mathsf{S}, a \in \mathsf{A}(s)\text{)} \end{aligned}$$

The choice of bounds. Any variable bounds that respect the unique solution will not change the answer. That is, any lb and ub with $0 \le lb(s) \le V(s) \le ub(s)$ yield a sound encoding. While these additional bounds are superfluous, they may significantly prune the search space. We investigate trivial bounds, e.g., knowing that all probabilities are in $[0,1]$, bounds from a structural analysis as discussed by [13], and bounds induced by a warm start of the solver. For the latter, if we have obtained values $V' \le V$, e.g., induced by a suboptimal policy, then $V'(s)$ is a lower bound on the value x_s, which is particularly relevant as the LP minimizes.

Equality for unique actions. Markov chains, i.e., MDPs where $|\mathsf{A}| = 1$, can be solved using linear equation systems. The LP encoding uses one-sided inequalities and the objective function to incorporate nondeterministic choices. We investigate adding constraints for all states with a unique action, i.e. for all $s \in S \setminus T$ with $|\mathsf{A}(s)| = 1$ we add (calling the unique action a in every state):

$$x_s \le \sum_{s' \in S \setminus T} \delta(s,a)(s') \cdot x_{s'} + \sum_{t \in T} \delta(s,a)(t)$$

These additional constraints may trigger different optimizations in a solver, e.g., some solvers use Gaussian elimination for variable elimination.

A simpler objective. The standard objective assures the solution η is optimal for *every* state, whereas most invocations require only optimality in some specific states – typically the initial state s_0 or the entry states of a strongly connected component. In that case, the objective may be simplified to optimize only the value for those states. This potentially allows for multiple optimal solutions: in terms of the MDP, it is no longer necessary to optimize the value for states that are not reached under the optimal policy.

Encoding the dual formulation. Encoding a dual formulation to the LP is interesting for mixed-integer extensions to the LP, relevant for computing, e.g., policies in POMDPs [65], or when computing minimal counterexamples [79]. For LPs, due to the strong duality, the internal representation in the solvers we investigated is (almost) equivalent and all solvers support both solving the primal and the dual representation. We therefore do not further consider constructing them.

3.2 How to Solve LPs with Existing Solvers?

We rely on the performance of state-of-the-art LP solvers. Many solvers have been developed and are still actively advanced, see [2] for a recent comparison on general benchmarks. We list the LP solvers that we consider for this work in Table 2. The columns summarize for each solver the type of license, whether it uses exact or floating-point arithmetic, whether it supports multi-threading, and what type of algorithms it implements. We also list whether the solver is available from the two model checkers used in this study[3].

Methods. We briefly explain the available methods and refer to [19] for a thorough treatment as well as to the introduction of [1] for recent developments. Broadly speaking, the LP solvers use one out of two families of methods. 1) *Simplex*-based methods rely on highly efficient pivot operations to consider vertices of the simplex of feasible solutions. Simplex can be executed either in the *primal* or *dual* fashion, which changes the direction of progress made by the algorithm. Our LP formulation has more constraints than variables, which generally means that the dual version is preferable. 2) *Interior methods*, often the subclass of *barrier methods*, do not need to follow the set of vertices. These methods may achieve polynomial time worst-case behaviour. Folklore claims that simplex has superior average-case performance but is highly sensitive to perturbations, while interior-point methods have a more robust performance.

Warm starts. LP-based model checking can be done using two types of warm starts. Either by providing a (feasible) basis point as done in [40] or by presenting bounds as discussed above. The former, however, comes with various remarks and limitations, such as the requirement to disable preprocessing. We therefore used warm starts only by using bounds.

Multithreading. We generally see two types of parallelisation in LP solvers. Some solvers support a *portfolio* approach that runs different approaches and finishes with the first one that yields a result. Other solvers parallelize the interior-point and/or simplex methods themselves.

Guarantees for numerical LP solvers. All LP solvers allow tweaking of various parameters, including *tolerances* to manage whether a point is considered feasible or optimal, respectively. The experiments on the M_n model of Example 2.5 shown in Table 1 already indicate that these guarantees are *not* absolute. We have investigated the influence of the *primal feasibility tolerance* and *dual feasibility tolerance* parameters of Gurobi and HiGHS as called from mcsta on the correctness using the same model. We chose Gurobi and HiGHS as they are the fastest and fastest open-source LP solvers, respectively, in our experimental evaluation in Section 6. The results are shown in Table 3. We set both tolerances to the same value indicated in the "tolerances" column. Gurobi accepts values in $[10^{-9}, 10^{-2}]$ and uses a default of 10^{-6}; HiGHS accepts values in $[10^{-10}, \infty)$ and uses a default of 10^{-7}. For each combination of tolerance values, solver, and property, we again write $n \leq \mathit{value}$ to report that this combination delivered correct results (i.e. within ± 0.01 of the correct value; wrong values were in fact always equal or very close to 0.5) for all values of n up to and including *value*.

Table 3: Correct results on M_n depending on tolerances

	Gurobi		HiGHS	
tolerances	P_{min}	P_{max}	P_{min}	P_{max}
10^{-4}	$n \leq 12$	$n \leq 12$	$n \leq 12$	$n \leq 12$
10^{-5}	$n \leq 15$	$n \leq 15$	$n \leq 15$	$n \leq 15$
10^{-6}	$\boldsymbol{n \leq 18}$	$\boldsymbol{n \leq 18}$	$n \leq 18$	$n \leq 18$
10^{-7}	$n \leq 22$	$n \leq 22$	$\boldsymbol{n \leq 22}$	$\boldsymbol{n \leq 22}$
10^{-8}	$n \leq 25$	$n \leq 25$	$n \leq 25$	$n \leq 25$
10^{-9}	$n \leq 28$	$n \leq 28$	$n \leq 28$	$n \leq 28$

We see that the correctness on the M_n model for these two solvers appears to depend only on the tolerance values, with no differences between the two solvers otherwise. The difference between Gurobi and HiGHS in Table 1 can be fully explained by the difference in their default tolerances (highlighted in bold in Table 3). For every tolerance setting, there is an n that causes incorrect results; given a required result precision, it is not possible to derive a model-independent tolerance that would guarantee correct results up to that precision.

Exact solving. SoPlex supports exact computations, with a Boost library wrapping GMP rationals [36], after a floating-point arithmetic-based startup phase [42]. While this combination is beneficial for performance in most settings, it leads to crashes for the numerically challenging models. Z3 supports only exact arithmetic (also wrapping GMP numbers with their own interface). We observe that the price of converting large rational numbers may be substantial. SMT solvers like Z3 use a simplex variation [30] tailored towards finding feasible points and in an incremental fashion, optimized for problems with a nontrivial Boolean structure. In contrast, our LP formulation is easily feasible and a pure conjunction.

[3] Support for Gurobi, GLPK, and Z3 was already available in Storm. Support for Glop was already available in mcsta. All other solver interfaces have been added for [53].

Table 2: Available LP solvers ("intr" = interior point)

solver	version	license	exact/fp	parallel	algorithms	mcsta	Storm
COPT [38]	7.1.3	academic	fp	yes	intr + simplex	yes	no
CPLEX [59]	22.1.1.0	academic	fp	yes	intr + simplex	yes	no
Gurobi [47]	11.0.0	academic	fp	yes	intr + simplex	yes	yes
GLPK [44]	5.0	GPL	fp	no	intr + simplex	no	yes
Glop [45]	9.10	Apache	fp	no	simplex only	yes	no
HiGHS [50,58]	1.7.0	MIT	fp	yes	intr + simplex	yes	no
lp_solve [17]	5.5.2.11	LGPL	fp	no	simplex only	yes	no
Mosek [73]	10.2	academic	fp	yes	intr + simplex	yes	no
SoPlex [43]	7.0.1	academic	both	no	simplex only	no	yes
Z3 [74]	4.13.0	MIT	exact	no	simplex only	no	yes

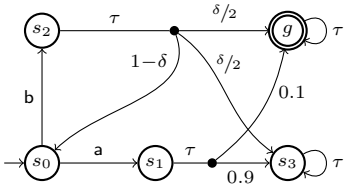


Fig. 3: Example MDP

4 Sound Policy Iteration

Starting with an initial policy, PI-based algorithms iteratively improve the policy based on the values obtained for the induced MC. The algorithm for solving the induced MC crucially affects the performance and accuracy of the overall approach. This section addresses possible correctness issues and some algorithmic choices.

4.1 Correctness of PI

The accuracy of PI is affected by the MC solver. Firstly, PI cannot be more precise than its underlying solver: the result of PI has the same precision as the result obtained for the final MC. Secondly, inaccuracies by the solver can hide policy improvements; this may lead to premature convergence with a sub-optimal policy.

Theorem 4.1 *When running Policy Iteration with an ε-precise solver for the induced Markov chains, the result of the overall algorithm can be off by more than ε.*

Proof Consider the MDP in Figure 3 with reachability objective $\mathrm{P}_{\max}(\{ g \})$. There is only one nondeterministic choice, namely in state s_0. The optimal policy is to pick b, obtaining a value of 0.5. Picking a only yields 0.1. However, when starting from the initial policy $\pi(s_0) = \mathsf{a}$, an ε-precise MC solver may return $0.1+\varepsilon$ for both s_0 and s_1 and $\delta/2 + (1-\delta) \cdot 0.1$ for s_2. This solution is indeed ε-precise. However, when evaluating which action to pick in s_0, we can choose δ such that a seems to obtain a higher value. Concretely, we require $\delta/2 + (1-\delta) \cdot 0.1 < 0.1 + \varepsilon$. For every $\varepsilon > 0$, this can be achieved by setting $\delta < 2.5 \cdot \varepsilon$. In this case, PI would terminate with the policy π_0 inducing a severely suboptimal value. In particular, for $\varepsilon < 0.4$, the result is not ε-precise. Note that the example can be modified to increase the distance between initial and optimal policy. □

If every Markov chain is solved precisely, PI is correct. Indeed, it suffices to be certain that one action is better than all others. This is the essence of modified policy iteration as described in [76, Chapters 6.5 and 7.2.6]. Similarly, [64, Section 4.2] suggests to use interval iteration when solving the system induced by the current policy and stopping when the under-approximation of one action is higher than the over-approximation of all other actions.

4.2 PI Implementations

Warm starts. PI profits from being provided a *good* initial policy. If the initial policy is already optimal, PI terminates after a single iteration. We can inform our choice of the initial policy by providing estimates for all states as computed by VI. For every state, we choose the action that is optimal according to the estimate. This is a good way to leverage VI's ability to quickly deliver good estimates [54], while at the same time providing the exactness guarantees of PI.

Markov chain solvers. To solve the induced MC, Storm can employ all linear equation solvers listed in [57] and all implemented variants of VI. We consider

- iterative solvers from GMM++ [39], i.e., the biconjugate gradient stabilised method (bigcstab), the generalized minimal residual method (gmres), and the quasi-minimal residual method (qmr), each potentially combined with a diagonal (diag) or an incomplete LU[4] (ilu) preconditioner,
- VI [23] with standard relative termination criterion,
- optimistic VI (OVI) [54], and
- the sparse LU decomposition implemented in Eigen [46] using either floating-point or exact arithmetic (LU^{X}).

LU and LU^{X} provide exact results (modulo floating-point errors in LU) while OVI yields ε-precise results. The other solvers do not provide any guarantees.

[4] LU refers to **L**ower unitriangular and **U**pper triangular decomposition of a matrix.

The topological approach discussed for MDPs in Section 2.4 can also be applied as an optimisation for the Markov chain solvers, since applying a strategy can change the underlying graph structure and hence the SCC-decomposition. Consequently, topological optimisations for PI can be applied in three different variants

- decompose the MDP into its SCCs and analyse them monolithically (topo+mono),
- decompose the induced MCs into their SCCs and analyse them (mono+topo),
- apply topological optimisations for both the MDP and the induced MCs (topo+topo).

In our experiments in Section 6.1.3 we analyse the performance of the different Markov chain solvers and topological variants.

5 Experimental setup

To understand the practical performance of the different algorithms, we performed an extensive experimental evaluation. Before we present its results in Section 6, we describe the experimental setup, in particular the selection of benchmark sets as well as the methods used for analyzing the collected data, in this section. Our data availability statement at the beginning of the paper links to an artifact containing all benchmarks and tools used as well as all logs created for this study.

5.1 Technical Setup

The bulk of our experiments were performed on Intel Xeon 8468 Sapphire systems running 64-bit Rocky Linux 8.9. We used the Slurm workload manager to allocate 4 CPU cores and 16 GB of RAM to each experiment. We explicitly note where deviate from this setup, notably in our analyses of the influence of threads in Section 6.1.2 and of hardware effects in Section 6.4.

5.2 Tool Configurations

A *tool configuration* results from choosing an algorithm (e.g. LP) and a specific variant thereof (e.g. the simplex algorithm) as well as all further hyper-parameters, including algorithm-specific optimizations (e.g. using equality constraints where possible), general optimizations (e.g. using topological methods), and technical decisions (e.g. the number of threads used, or whether we use Storm's or mcsta's implementation of the algorithm and optimizations).

The number of tool configurations is exponential in the number of hyper-parameters; given the parameters that we consider, it is infeasible to run all possible configurations on all benchmarks. We therefore investigate each hyper-parameter separately, choosing reasonable values for the others that are determined based on other experiments. Every subsection of Section 6 will make it clear which algorithms and hyper-parameters it considers. Throughout the experiments, the model checker used in a specific configuration is indicated by the subscripts $_{\mathsf{s}}$ for Storm and $_{\mathsf{m}}$ for mcsta.

5.3 Benchmark Sets

An evaluation depends heavily on the selected benchmarks. To make this dependency transparent, we consider several benchmark sets, each of which consists of a number of *benchmark instances*: combinations of a model, a valuation for the model's parameters, and an objective to analyse. In the following, we first describe the base sets, i.e. the origin of our benchmark instances. These are *qvbs*, *mecs*, and *gridworlds*. All three of these sets together form the *alljani* set, named for the fact that all its instances are available in the JANI modelling language [21]. From them, we select a representative subset, the *practitioner-set-2024*. Finally, we describe the separate *premise* set.

qvbs set. These are all benchmark instances from the Quantitative Verification Benchmark Set (QVBS) [55] that satisfy the following basic criteria: They have an MDP, Markov automaton (MA), or probabilistic timed automaton (PTA) model[5] and a reachability or expected reward/time objective that is quantitative, i.e. not a query that yields a zero or one probability. We only consider instances where both Storm and mcsta can build the explicit representation of the MDP within 5 minutes. This yields 366 instances. We obtain reference results for 341 of them from either the QVBS database or by using one of Storm's exact methods. The reference results obtained via different methods are consistent.

gridworlds set. This is a set of gridworlds that have been previously investigated and that were colloquially known to be hard. We collected 2-dimensional grid worlds with and without moving obstacles from a gridworld repository with partially observable MDPs originally presented in [60] and took the underlying (fully observable) MDP.[6] We additionally take a gridworld from [61] that differs from the other benchmarks in the movement dynamics, and a random walk model that is interesting for its

[5] MA and PTA are converted to MDP via embedding and digital clocks [66].

[6] Notice that these benchmarks are partially similar to the original MDPs from [61] *before* their (significant) transformation that yields the *premise* set.

concise representation of N-dimensional random walks. All benchmarks are scalable along multiple dimensions. We include 27 instances that compute the maximal probability to reach a specific goal and 20 instances for the minimal expected number of steps until a target state is reached. For 30 out of those 47 instances, a reference result was obtained using one of Storm's exact methods. Most instances include non-trivial MECs, but within this benchmark set, only the maximal probability instances have MECs that affect convergence.

mecs set. This is a set of models explicitly collected to include MDPs containing non-trivial (i.e. not target, sink, or single-state) MECs. This is important because many algorithms explicitly include methods for dealing with them, but neither *qvbs* nor *premise* contain MDPs with non-trivial MECs. The set *mecs* includes the instances from *gridworlds* exhibiting MECs, namely the *avoid-MDP* and *evade-MDP* models. Additionally, we include the models *sensors*, *mer* [33], and the handcrafted extreme cases *BigMEC* and *MulMEC* (turned into MDPs by merging the players of their original stochastic game formulations) from [64]. Moreover, we extended the two *qvbs* models *consensus* and *wlan* with MECs by putting the original models' behaviour in parallel composition with an (ineffective) looping structure that, in each step and with uniform probability, sets a variable x to either $x + 1 \mod 10$ or 0. All in all, the *mecs* contains 36 instances: 21 with maximum probability and 15 with minimum expected reward objectives. A reference result is available for 24 instances.

practitioner-set-2024. This benchmark set is a selection of instances from the *alljani* set chosen to be

- *concise*: we want the set to contain a limited number of instances so that evaluating different configurations of an algorithm on this set remains feasible,
- *diverse*: we want diversity with respect to model structure,
- *non-trivial*: we want non-trivial runtimes that are robust to noise but fast enough to ensure a reasonable total runtime, and
- *representative*: we want the set to be representative of the whole set of benchmarks that we are aware of.

We used the following selection process: First, to obtain data on how easy or hard benchmarks are for certain algorithms, we ran a subset of algorithms on all instances of the *alljani* set. We in particular ran at least one variant of each class of algorithm (naive and sound VI, PI, and LP), namely the one that performed best in the conference version of this paper [53]. Then, we restricted to the benchmark instances where at least one algorithm took more than 5 seconds (so that instances are non-trivial) and at least one algorithm succeeded in less than 120 seconds (to exclude models that take too long, supporting conciseness). The resulting set contained 144 benchmark instances.

We selected 80 benchmark instances such that evaluating one algorithm on all of them is expected to take less than 2 hours. To arrive at this number, we manually removed benchmark instances to reduce the size of the set and to avoid certain models for which many parameter valuations survived our initial filtering[7] being overrepresented. Hence, we included 12 instances from *mecs*, ensuring that these relevant structures are present. Further, we selected 50 instances from *qvbs*, thus including structures appearing in the more realistic case studies. The final 18 instances are from *gridworlds*. Additionally, we ensured that the set is balanced between reachability and total reward objectives (35 reachability, 45 reward). 71 instances have a reference result.

premise set. This is a set of numerically challenging models from a runtime monitoring application [61], named for the corresponding prototype. The original problem is to compute the normalized risk of continuing to operate the system being monitored subject to stochastic noise, unobservable and uncontrollable nondeterminism, and partial state observations. This is a query for a conditional probability, answered via probabilistic model checking by unrolling an MDP model along an observed history trace of length $n \in \{ 50, \ldots, 1000 \}$ following the approach of Baier et al. [12]. The MDPs contain many transitions back to the initial state, ultimately resulting in numerically challenging instances (containing structures similar to the one of M_n in Section 2.3). We were able to compute a reference result for all 200 instances.

We separately analyse the *premise* set and do not include it in the *practitioner-set-2024* because, due to their origin, the MDPs of the *premise* set are explicitly represented while we wanted the *practitioner-set-2024* to include only models specified in JANI for the purpose of cross-tool compatibility.

5.4 Data Analysis

Collected data. For each run of an algorithm on a benchmark instance, we save the runtime. The runtime includes necessary and/or stated preprocessing, but *not* the time for constructing the MDP state space (which is independent of the algorithms). mcsta reports all time measurements rounded to multiples of 0.1 s. We summarize timeouts, out-of-memory failures, errors, and

[7] For example, there were 10 instances of the *zeroconf* model that satisfied the initial filter. We reduced this to 5 instances by picking at least one representative of every model-objective combination, but using only a subset of the parameter values, typically the largest and smallest.

incorrect results as "n/a". Throughout most of our evaluation, the timeout is 15 minutes for total runtime including MDP construction. For the hardware and LP tolerances experiments, we reduced the timeout to 10 minutes.

We define *incorrectness* as follows: A result $\bar{v}$ is incorrect if $|v - \bar{v}| > v \cdot 10^{-3}$ (i.e. a relative error of $\pm 10^{-3}$) whenever a reference result v is available. We however do not flag a result as incorrect if v and $\bar{v}$ are both below 10^{-8} (relevant for the *premise* set), except in the LP tolerances experiment. Nevertheless, we configure the (unsound) convergence threshold for VI as 10^{-6} relative; among the sound VI algorithms, we include OVI, with a (sound) stopping criterion of relative 10^{-6} error. To only achieve the 10^{-3} precision we actually test, OVI could thus be even faster than it appears in our plots. We make this difference to account for the fact that many algorithms, including the LP solvers, do not have a sound error criterion. We mark exact algorithms/solvers that use rational arithmetic with a superscript $^{\text{X}}$. The other configurations use floating-point arithmetic (fp).

Data analysis methods. To compare the algorithms, we use several methods, each of which has its own strengths and weaknesses.

- *Quantile plots* (e.g. top left of Fig. 4) compare multiple tool configurations; for each, we sort the instances by runtime and plot the corresponding monotonically increasing line. Here, a point (x, y) on the a-line means that the x-th fastest instance solved by a took y seconds. These plots provide an overview of algorithm performance on the whole set, but hide performance on separate instances due to ordering them for each algorithm separately.
- *Scatter plots* compare two tools or configurations (e.g. top right of Fig. 4; note that only this scatter plot contains the legend indicating what the shapes and colours of the points mean). Each point (x, y) is for one benchmark instance: the x-axis tool took x while the y-axis tool took y seconds to solve it. The shape of points indicates the model type; the mapping from shapes to types is the same for all scatter plots and is only given explicitly in the first one in Fig. 6. The dashed lines around the solid line indicate the points where one tool is exactly twice as fast as the other. Intuitively, if most points are above the diagonal, the x-axis tool configuration is faster, and dually the y-axis tool configuration if most points are below the diagonal. These plots allow for comparing the performance on individual benchmark instances, but if there are many points both above and below the diagonal, it is difficult to draw conclusions. Further, they can only compare two tool configurations and are thus not suitable for getting an overview of larger numbers of configurations.

Note that the y-axis of both the quantile and scatter plots is logarithmically scaled. This allows assessing the relative performance of the algorithms for different benchmark sizes. Further, we highlight that the scatter plots depict all runtimes between 512 seconds and our timeout of 15 minutes (900 seconds) on the ≥ 512 line. Benchmarks on this line are solved (albeit in a long time), in stark contrast to the n/a lines where the result was incorrect or an error happened.

Finally, for clarity and succinctness of presentation, we do not show all plots which a reader might be interested in, restricting to those that we consider most relevant. We refer to our artifact (see the data availability statement at the beginning of the paper) for the full logs of our evaluation which allow for further analysis.

6 Experimental evaluation

In this section, we provide the results of our experimental evaluation. First, in Section 6.1, we individually study all hyper-parameters we consider, looking into the variants of each class of algorithm (VI, LP, and PI) separately, and then discussing the optimizations described in Section 2.4. We then investigate factors other than algorithm selection that can affect the result of an empirical comparison, namely benchmark selection (Section 6.2), state ordering (Section 6.3), and differences in hardware (Section 6.4). Finally, Section 6.5 discusses all the results and provides clear take-away messages from our extensive analysis.

6.1 Hyper-Parameter Analysis

This subsection analyses many different tool configuration (cf Section 5.2). To make this feasible, we restrict to the 80 instances of the *practitioner-set-2024* (cf. Section 5.3). Using this more structurally diverse set is an improvement over the conference version of this paper [53] which used the *qvbs* set.

6.1.1 Value Iteration

VI variants. The quantile plot in Fig. 4 shows the overall results for the different variants of the algorithm, namely standard VI, optimistic VI [54], sound VI [77], interval iteration [49] and the exact variant of VI called rational search [69]. We tested two implementations of each algorithm, one in Storm (lighter colour) and one in mcsta (darker colour), except for rational search which mcsta does not implement.

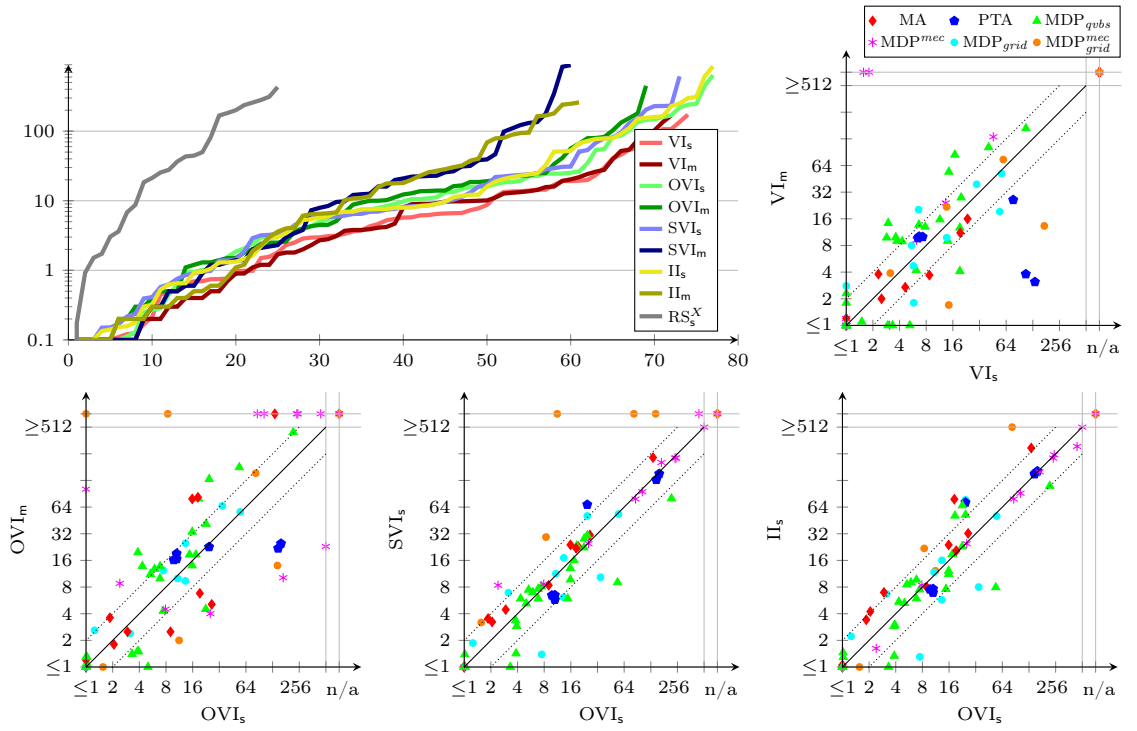


Fig. 4: Comparison of VI-based methods.

In terms of **performance**, naive VI is the fastest while the rational search is the slowest, the latter only being able to solve 25 instances within the given time. For the VI variants that provide ε-precise results and use floating-point arithmetic, there is no clear winner. OVI$_s$ and II$_s$ solve the most instances (77), but the scatter plots show that there are benchmark instances where other algorithm variants are preferable. In particular, the implementation of algorithms (VI$_s$ vs. VI$_m$ or OVI$_s$ vs. OVI$_m$) has an impact that is comparable to that of switching the algorithm variant (OVI$_s$ vs. SVI$_s$ and OVI$_s$ vs. II$_s$). This adds a nuance to claims in [54,77] where OVI and SVI seemed clearly preferable to II; possibly, those results were due to inefficient implementations or due to a different selection of benchmarks.

In terms of **reliability**, naive VI produced 5 and 7 wrong results in Storm and mcsta, respectively. OVI, SVI and II had no incorrect results in Storm, but 2, 13, and 12, respectively, in mcsta[8]. Of course, RS^X_s was always correct.

Optimizations. We discuss the effects of the commonly implemented optimizations described in Section 2.4. The top row of Fig. 5 shows the effect of enabling the **qualitative precomputations** in mcsta[9]. Quite surprisingly, the precomputations appear to consistently worsen the performance on some benchmark instances.

The second row of Fig. 5 shows the effect of using the **topological optimization** (which, for VI algorithms, only Storm offers). For OVI and RS^X, it significantly improves the performance on many benchmarks. Interestingly, for VI, the scatter plot is more mixed with several benchmark instances from the *mec* and *gridworld* sets being more than 30 times slower. We conjecture that Storm's implementation is inefficient in handling these models because they contain many (small) SCCs.

The last row of Fig. 5 shows the effect of using **MEC collapsing**. Recall from Section 2.4 that collapsing MECs is necessary for the convergence of the sound VI algorithms. We see this in the bottom right scatter plot: several *mecs* benchmarks do not terminate using OVI, but do when using OVI-mec. The RS^X algorithm always uses MEC-collapsing for this reason. The overhead for searching MECs can (mostly negatively) affect the runtime on instances that do not exhibit any (non-trivial) MECs, as is visible by all points in the scatter plots that are not from the *mecs* $*$ or *gridworld-mecs* $\otimes$ sets. Interestingly, MEC collapsing in Storm slows down the solving of several instances from the *mecs* set, while in mcsta, both VI and OVI generally benefit from this preprocessing if the models contain MECs.

[8] We believe this to be at least partly due to bugs in mcsta's MEC-collapsing implementation that we are working to fix.

[9] Recall that Storm does not allow turning these precomputations off.

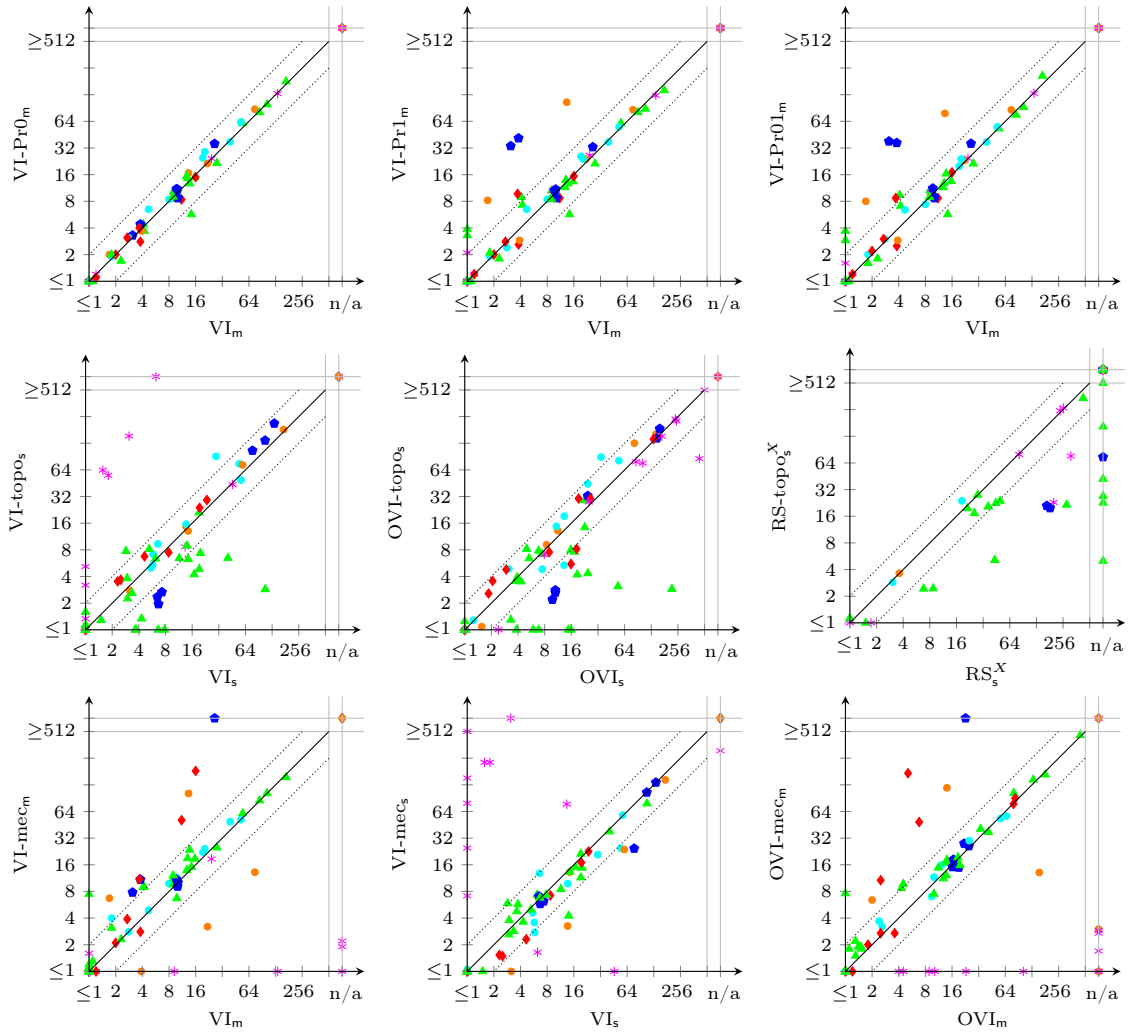


Fig. 5: Comparison of optimizations on different VI-based methods and implementations.

Table 4: LP summary

solver	correct	incorr.	no result
$\mathsf{VI_s}$	74	5	1
$\mathsf{VI_m}$	72	7	1
$\mathsf{COPT_m}$	44	1	35
$\mathsf{CPLEX_m}$	28	1	51
$\mathsf{Glop_m}$	19	0	61
$\mathsf{GLPK_s}$	7	0	73
$\mathsf{Gurobi_s}$	47	1	32
$\mathsf{Gurobi_m}$	45	1	34
$\mathsf{HiGHS_m}$	38	1	41
$\mathsf{lp\_solve_m}$	6	0	74
$\mathsf{Mosek_m}$	32	0	48
$\mathsf{SoPlex_s}$	11	1	68
$\mathsf{SoPlex_s^X}$	10	0	70
$\mathsf{Z3_s^X}$	1	0	79

6.1.2 Linear Programming

LP solvers. First, we compare different LP solvers. We apply no optimizations or reductions to the MDPs except for the precomputation of probability-0 states (and in Storm also of probability-1 states), and use the default settings for all solvers, with the trivial variable bounds $[0, 1]$ and $[0, \infty)$ for probabilities and expected rewards, respectively. We include VI as a baseline. We summarize the results in Table 4 and Fig. 6 (left).

In terms of **performance** and scalability, Gurobi solves the highest number of benchmarks in any given time budget, closely followed by COPT. CPLEX, HiGHS, and Mosek make up a middle-class group. While the exact solver Z3 is very slow, SoPlex's exact mode actually competes with some fp solvers. However, the quantile plots do not tell the whole story: On the right of Fig. 6, we compare COPT and Gurobi directly and see that each

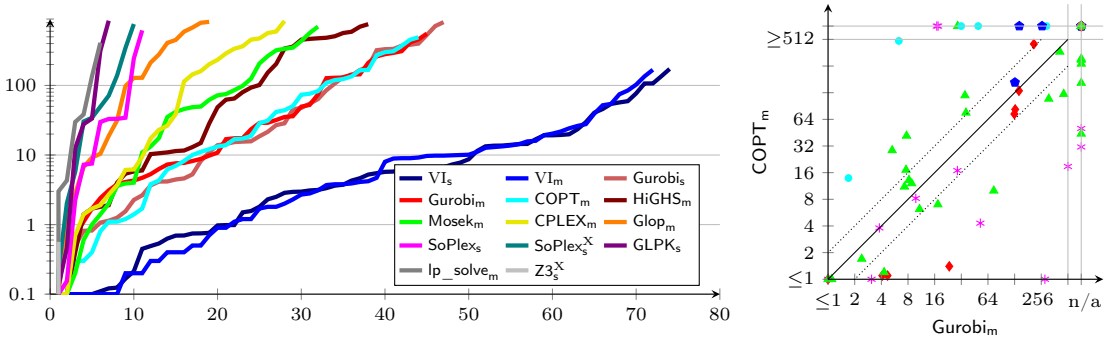


Fig. 6: Comparison of LP solver runtime on the *community* set.

has a large number of instances on which it is (much) better.

In terms of the **reliability** of results, most solvers produce exactly one incorrect result on the *practitioner-set-2024*. This number is smaller than in Table 3 of the conference version of this paper [53] because we considered a different and larger (but arguably less diverse) set of benchmark instances for the same evaluation in that paper. For more discussion of the (lack of) reliability of LP solvers, see below and Sections 2.3 and 3.2.

Overall, Gurobi achieves the highest performance at decent reliability; in the remainder of this section, we thus use Gurobi_s whenever we apply non-exact LP.

LP feasibility tolerances. In Section 3.2, we studied the impact of changing the LP solver's default primal and dual feasibility tolerance values on the artificial M_n model using Gurobi and HiGHS. To better understand how these parameters influence the reliability of results in a more practical setting, we also ran Gurobi and HiGHS with different tolerance settings on the *practitioner-set-2024* set. We again set both tolerances to the same value, evaluating values 10^{-9} as the "most precise" configuration (with 10^{-9} being the lowest value Gurobi accepts), 10^{-7} as it is the default for HiGHS, 10^{-6} as it is the default of Gurobi, 10^{-4}, and 10^{-3} as the "least precise" setting where the tolerance value matches the relative error we require. We use the topological optimization.

The results are summarised in Table 5. We again[10] list the number of correct and incorrect results, but differentiate the "no result" case into errors (where solving the LP failed, e.g. due to running out of memory or because the solver considers the problem infeasible) and timeouts (with a timeout of 10 minutes here). Column "time" lists the total time in minutes, including timeouts; it can only serve as a rough indication for how the tolerance influences the runtime.

We see that lower tolerances in general lead to fewer incorrect results but more timeouts and longer runtimes, with value 10^{-9} being sufficient to avoid all incorrect results on this benchmark set (unless instances that timed out would, given enough time, lead to an incorrect result). However, the behaviour is not monotonic: for example, Gurobi shows one timeout more for 10^{-4} than for all tighter tolerances, and HiGHS delivers two incorrect results for 10^{-7} but none for both 10^{-6} and 10^{-9} (these may turn into errors and/or timeouts).

Configuration. Gurobi can be configured to use an "auto" portfolio approach, potentially running multiple algorithms concurrently on multiple threads, a primal or a dual simplex algorithm, or a barrier method algorithm. We compared each option (using 4 threads) and summarize the results in Figure 7. The quantile plot shows that, overall, there is no huge difference in performance. The auto configuration is usually comparable to the dual and barrier algorithm, but in several cases outperforms them by a factor of more than 2. On the other hand, the primal algorithm is the only configuration that similarly outperforms auto on several benchmark instances, but is also often significantly slower. Interestingly, in the experiments for the conference version of this paper [53], we had found that on the whole *qvbs* set, primal solves the fewest instances, whereas we now find that on the *practitioner-set-2024*, it solves the most (50, compared to the 49 of auto). Nonetheless, based on the previous experiments and its very similar performance to primal, we select the auto configuration (the default of Gurobi) for the following experiments.

We highlight the scatter plot in the top-right of Figure 7 comparing the dual simplex configuration to the barrier method. The barrier method has polynomial worst-case complexity, unlike the potentially exponential simplex method. In this experiment, they perform

[10] Note the the numbers in Table 5 are not comparable with those in Table 4 because the two experiments ran on different hardware, so timeouts in the data for one table may show up as any type of entry in the other table.

Table 5: The impact of LP feasibility tolerance values on the *practitioner-set-2024* set.

	Gurobi					HiGHS				
tolerances	correct	incorr.	error	timeout	time (min)	correct	incorr.	error	timeout	time (min)
10^{-3}	54	9	3	14	209	48	8	5	19	250
10^{-4}	52	8	3	17	228	49	8	4	19	250
10^{-6}	60	1	3	16	220	56	0	4	20	259
10^{-7}	60	1	3	16	222	56	2	1	21	278
10^{-9}	61	0	3	16	225	56	0	2	22	281

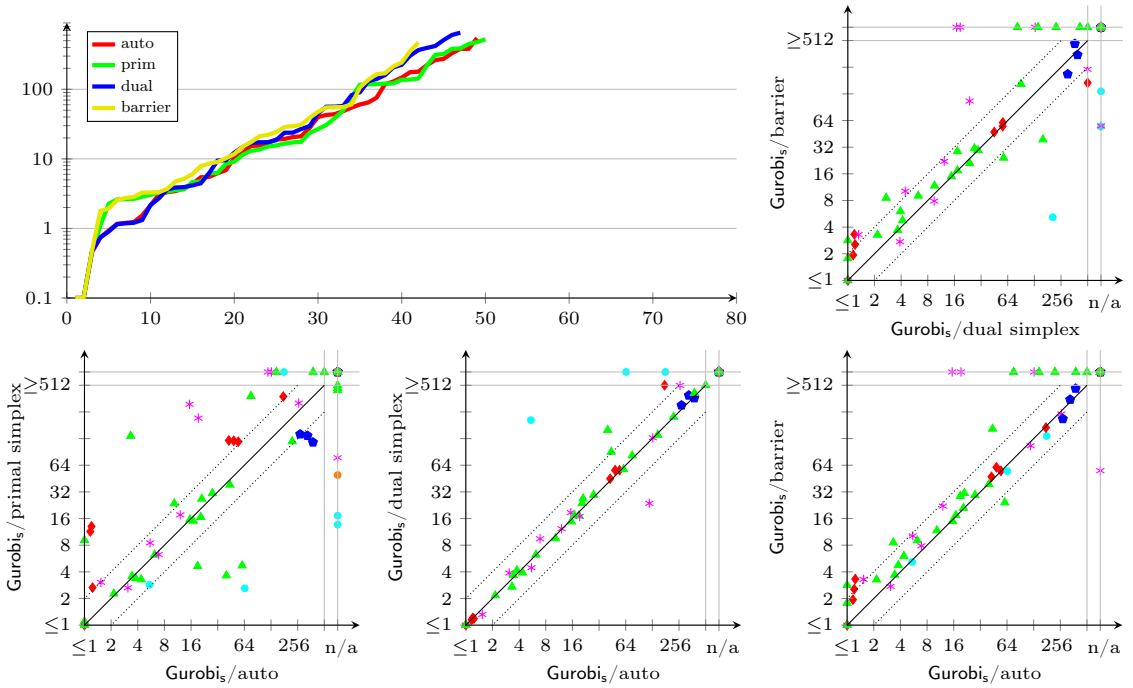


Fig. 7: Comparison of Gurobi's configurations.

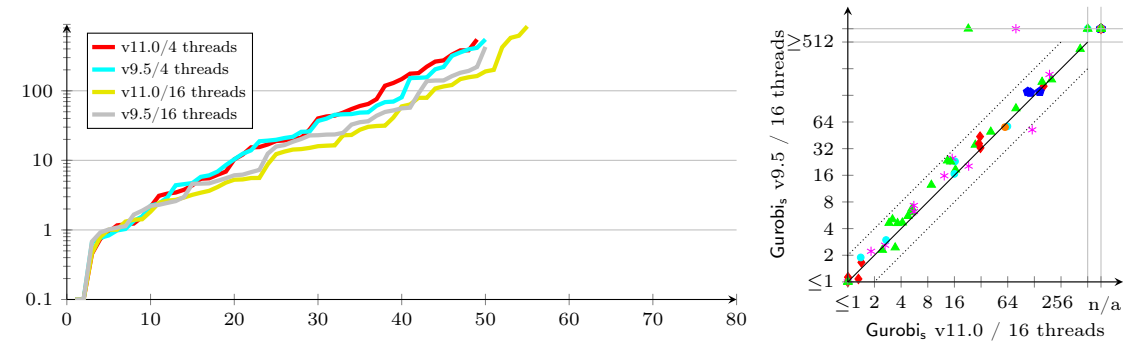


Fig. 8: Comparison of Gurobi v11.0 (Nov 2023) and v9.5 (Nov 2021).

comparably on most benchmarks, although the simplex method solves more benchmark instances (47 versus 42).

Solver version. To also assess the extent of improvements from updating the underlying solver, in Fig. 8 we compare Gurobi's version 9.5 (November 2021, used in [53]) and version 11.0 (November 2023, used in this paper). We test both versions with 4 threads and 16 threads.

With 16 threads, version 11 is slightly faster and solves 56 benchmarks versus 51 with version 9.5. With 4 threads (the default setting in the rest of the paper), version 9.5 seems to be slightly faster.

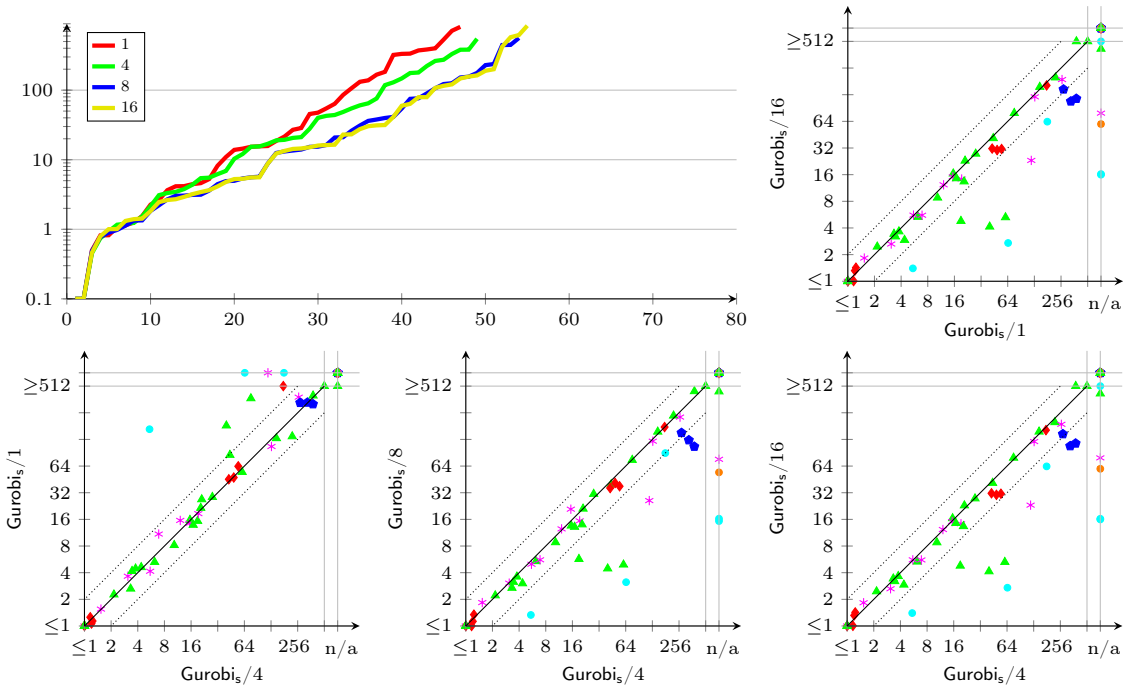


Fig. 9: Comparison of how the number of threads affect the performance of Gurobi's auto method.

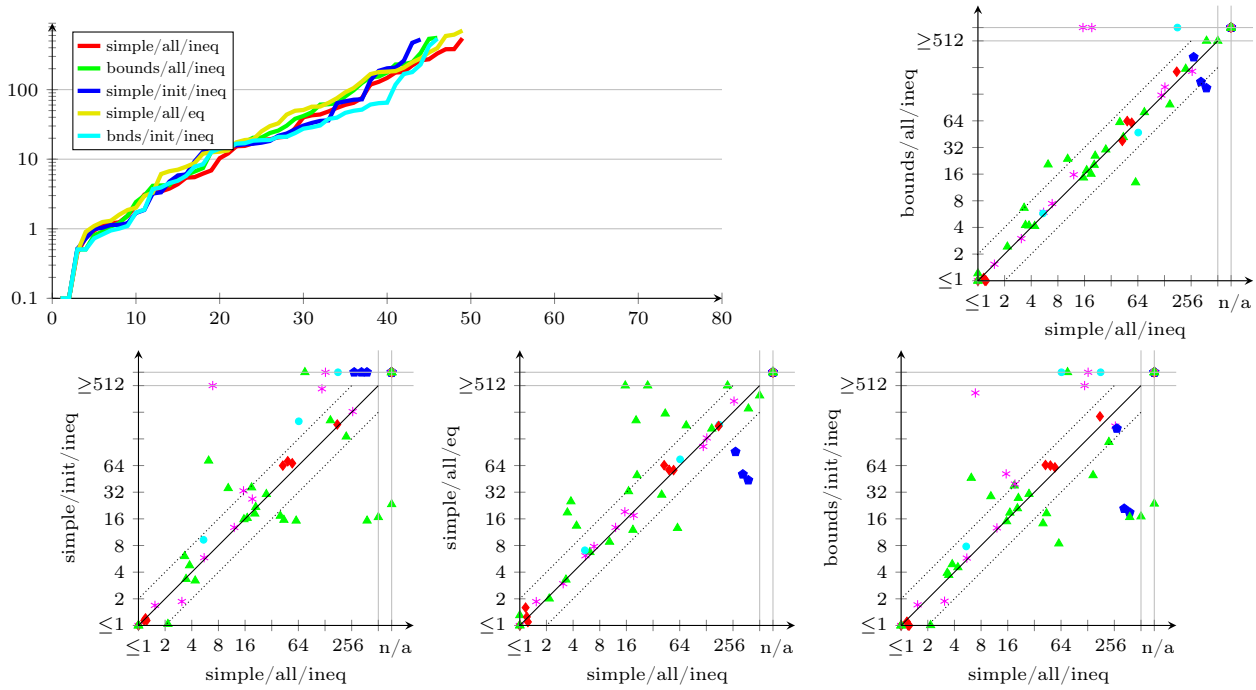


Fig. 10: Performance impact of LP problem formulation variants (using Gurobi$_s$ with 4 threads)

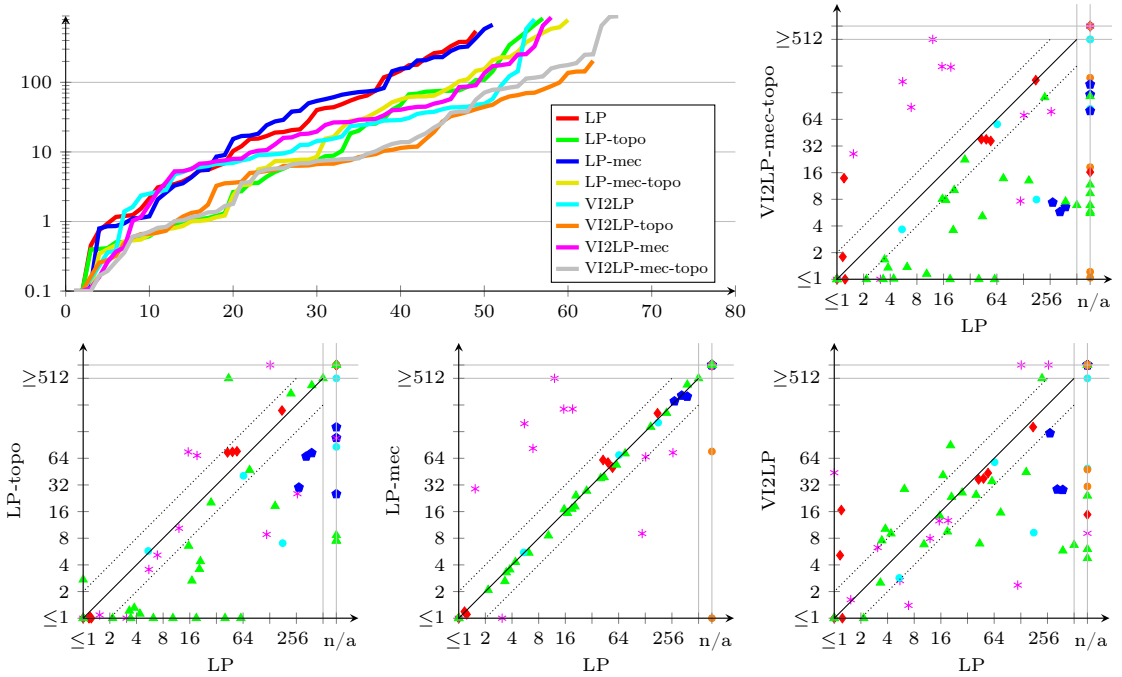


Fig. 11: Comparison of the common optimizations for LP algorithms, using Gurobi$_s$ v11.0, 4 threads and auto.

Threads. In Fig. 9, we more specifically investigate how the number of threads affect Gurobi's performance. In contrast to the results in [53] with Gurobi's version 9.5, where the number of threads had no significant impact, here we see that 8 and 16 threads are sometimes significantly faster than 1 or 4 threads and solve several instances more (49 with 4 threads vs. 55 with 16 threads). Nonetheless, in all following experiments, we use 4 threads unless noted otherwise in order to allow for reasonable parallelization of our experiments.

LP formulation. Fig. 10 shows the performance impact of modifying the LP formulation by either

- supplying Gurobi with more precise bounds on the variables for expected reward objectives using methods from [13,70] ("bounds" instead of "simple"),
- optimizing only for the initial state ("init") instead of for the sum over all states ("all"), or
- using equality ("eq") instead of less-/greater-than-or-equal ("ineq") constraints for unique action states.

As in [53], no LP formulation is clearly superior, but each modification may make a significant difference.

Optimizations. Fig. 11 shows that optimizations from Section 2.4 have a significant positive impact overall. The warm start and topological optimizations generally reduce runtime (with few exceptions) and increase the number of solved instances. The effect of MEC collapsing is more mixed: it often increases runtime, but it also increases the number of solved instances. Likely, the overhead of Storm's MEC-collapsing discussed in Section 6.1.1 is also relevant here. Overall, we recommend using all the common optimizations.

6.1.3 Policy Iteration

Solver for the induced MC. For PI, the main hyperparameter is the choice of the solver for the induced MC. The quantile plot in Fig. 12 shows that gmres with the incomplete LU precondition (gmres-ilu) usually performs best, being fastest and solving the most instances. The scatter plots comparing it to its closest competitors show that all of them (bigcstab-ilu, qmr-ilu and VI) also outperform gmres-ilu on some instances. Following Theorem 4.1, only PI/LU^X is guaranteed to produce the correct result. Interestingly, it solves more instances than when using inexact LU or OVI as solver for the induced MC, potentially because floating-point errors induce spurious policy changes. Most of the other tool configurations considered in Fig. 12 indeed produce several incorrect results, namely 3 (VI, qmr), 7 (gmres-ilu), and 10 (gmres-diag, gmres-ilu and bigcstab-ilu). OVI and LU produce no incorrect results on the *practitioner-set-2024*.

Optimizations. Fig. 13 shows the impact of the topological optimizations and warm starts. We do not test qualitative precomputations (as these are always on in Storm while mcsta does not support PI) and MEC collapsing (which is not implemented for PI in Storm).

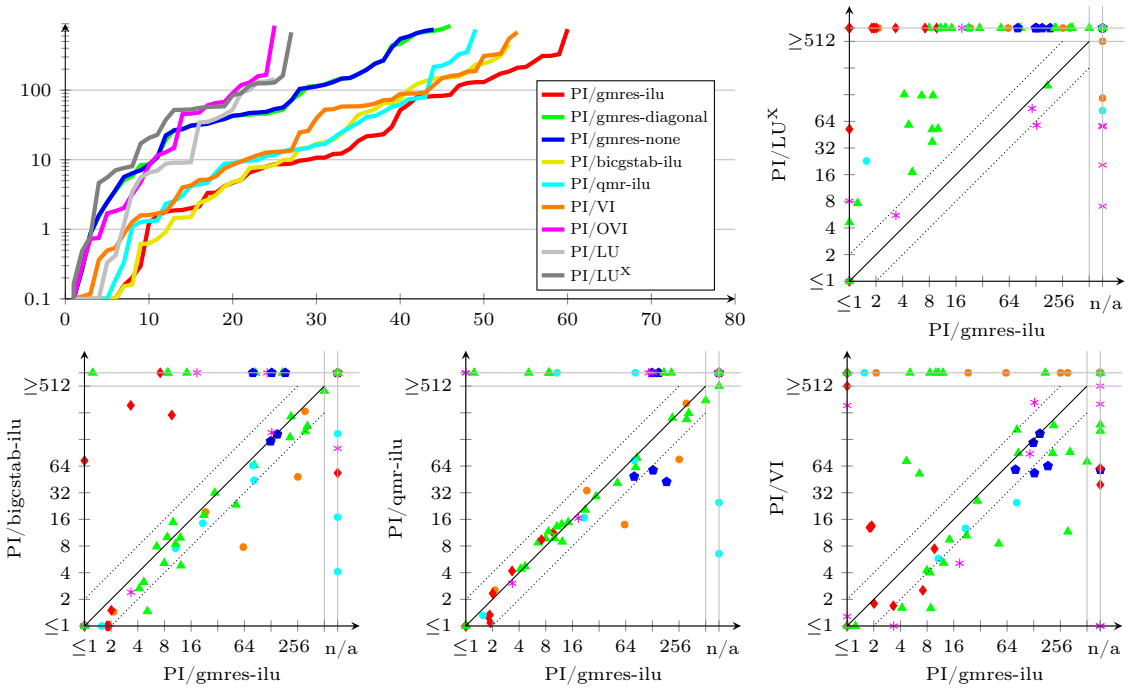


Fig. 12: Comparison of different algorithms for solving induced Markov chains in PI (without optimizations).

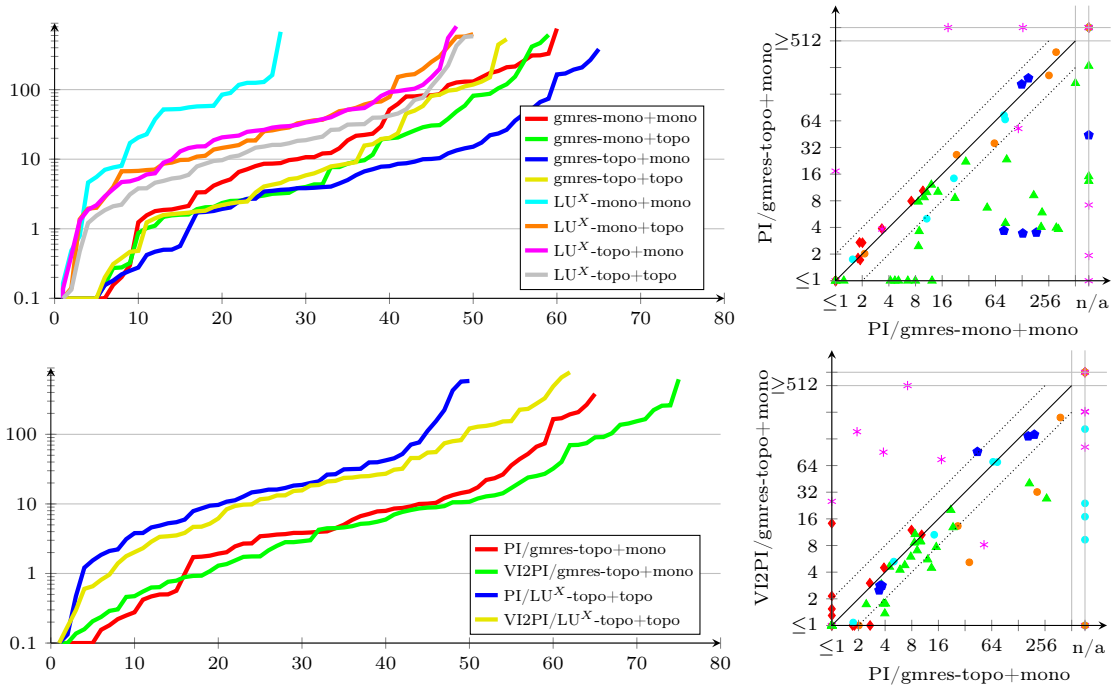


Fig. 13: Comparison of PI optimizations: topological solving (top) and warm starts (bottom).

For the **topological optimizations**, we use the notation introduced in Section 4.2, distinguishing the levels at which they can be applied. Trying all 4 combinations of monolithic and topological solving with gmres-ilu, we conclude that topo/mono is best, i.e. decomposing the MDP into SCCs, but solving the induced MC monolithicly. The scatter plot shows that this holds for essentially all instances. For LU^X, mono+mono is significantly worse than the tool configurations with topological optimizations, solving only 27 instances whereas the other 3 solve 48 (topo+mono) or 50 (mono+topo and topo+topo). Overall, topological optimizations significantly improve the performance of PI.

For the **warm start** optimization, we use VI to compute an initial policy. We denote this tool configuration VI2PI. Applying warm starts on the best floating point variant gmres-topo+mono and the best exact variant LU^X-topo+topo yields a significant increase in solved instances for both methods. The scatter plot shows that this optimization is also beneficial on almost all instances.

6.1.4 Optimizations

Summarizing the previous discussions, we conclude the following:

- An optimization does not necessarily improve the performance of an algorithm on all instances. Instead, it can either show mixed results, as for the different LP formulations, or even be detrimental, like the qualitative precomputations of mcsta for VI.
- Generally, it appears favourable to use topological methods and warm starts, as they clearly improved the performance of PI and LP as well as of the sound VI variants.
- The implementation of an optimization naturally affects its performance, as visible in the difference between MEC collapsing for VI in mcsta vs. Storm.

We further experimented with the bisimulation and essential states optimisations, which we did not consider so far.

Bisimulation. Fig. 14 shows the effect of first reducing the MDP by computing the bisimilar states (available in Storm) and then solving it. We used monolithic naive VI as inexact and VI2PI-topo+topo as exact solution method. Clearly, for almost all benchmark instances in the *practitioner-set-2024*, the overhead for computing the bisimulation is not worth the reduction in runtime for working on a smaller MDP.

Essential states. Fig. 15 shows the effect of reducing the MDP to only the essential states. This reduction is implemented in mcsta, and we tested it for VI. The results appear promising: Using the reduction significantly lowers the runtime on many benchmark instances, albeit at the cost of solving one less instance.

6.2 Influence of the Benchmark Set

We now show how the selection of benchmarks greatly affects the performance of algorithms, and hence also the conclusions one can draw from an empirical comparison. For this, we show the performance of a selection of algorithms on the *practitioner-set-2024*, the *alljani* set, the *hard* set and the *premise* set (see Section 5.3 for a description of these sets) in Fig. 16.

On the *practitioner-set-2024*, we list more algorithms, since on this small set it is feasible to run them all (as we did for the analysis in Section 6.1 above). For the other sets, we only consider a subset of tool configurations, using one floating-point and one exact representative for each class of algorithm. We always use Storm and the topological optimization, and pick (i) VI as baseline, (ii) OVI and RS^X as representatives for VI, (iii) PI with warm start and gmres or LU^X as solver for the induced MC, and (iv) LP with warm start and MEC-collapsing, solved by Gurobi with 4 threads and the auto configuration or by the exact variant of SoPlex.

Additionally, we include two lines called "best" and "sel-best" in Fig. 16. The former plots the runtime resulting from, on every instance, selecting the optimal algorithm for this instance. With the latter, we try to approximate this line by using a selection among few tool configurations, namely VI-topo-mecq_s, PI/gmres-mono+topo_s, VI-es_m and II_m. These configurations were selected by choosing those that have near-optimal runtime on most of the benchmarks in the *practitioner-set-2024*. Note that sel-best is not given for the *premise* set, as we did not run all relevant configurations on this set, and because VI is always expected to perform poorly on this particular set. Similarly, this selection is not applicable for the exact solvers on the right of Fig. 16.

Given this evaluation on multiple benchmark sets, we can draw conclusions about what the best solution method for MDP model checking may be, and how this is influenced by the benchmark set—leaving it to the practitioner to determine which of our benchmarks the models resulting from their concrete case study are most similar to.

Best solution methods. We thoroughly investigated the state of the art in MDP model checking, showing that there is no single best algorithm for this task. Overall, although LP has the superior (polynomial) theoreti-

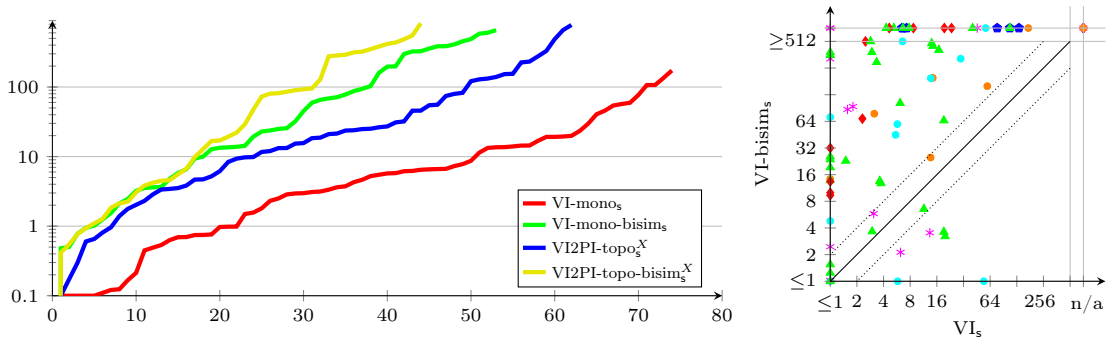


Fig. 14: Impact of Bisimulation minimization

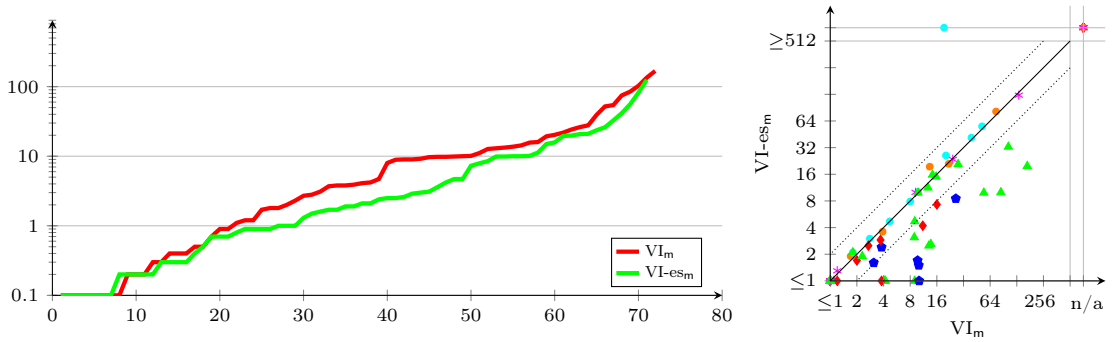


Fig. 15: Impact of essential state optimization

cal complexity, in our practical evaluation, it almost always performs worse than the other (exponential) approaches. This is even though we use modern commercial solvers and tune both the LP encoding of the problem as well as the solvers' parameters. In general, OVI and VI2PI are performing well on the *alljani* set, and on the *practitioner-set-2024* in particular. This confirms the observations made in [53] on the *qvbs* and *hard* set, both of which are subsets of the *alljani* set. Still, both methods are far from the best line in Fig. 16. Thus, we conclude that to achieve good performance on all kinds of instances, a selection (such as the one used for the sel-best line) of algorithms is necessary. An interesting open question is to find heuristics to recommend a method based on constructing and analyzing the state space of the instance.

We highlight again that using floating-point methods always carries the risk of incorrect results, even when using theoretically exact methods like LP and PI.

Impact of benchmark selection. On the one hand, there are benchmark sets that are very focussed on instances of a particular type. Concretely, on the *premise* set, VI based methods are clearly worse than those based on PI or LP. In particular, VI is wrong on many instances, as they are numerically challenging. On the other hand, Fig. 16 suggests that we succeeded in selecting a representative subset of *alljani* with the *practitioner-set-2024*, as the overall trends are similar in both plots. Note that we omitted the plots for the *hard* subset described in [53, Sec. 5.2], since they look essentially the same as the one for *alljani*. This suggests that our additions to the *qvbs* set have made the benchmark set more diverse, not favouring VI-based methods any more.

6.3 Influence of the State Ordering

We investigate the impact of permuting the internal order of the MDP's states: While states are mathematically a set, the transition matrices order these states. The representation of this matrix is relevant. In particular, all implementations of VI we consider use Gauss-Seidel VI, i.e. every update to the estimate vector happens in-place. Similarly, when solving the induced MC in PI, linear equation solvers like gmres can take advantage of the equation system's structure.

In Fig. 17, we show the effect of permuting the states for all three classes of algorithms. We use standard VI, PI with gmres, and LP with Gurobi, all implemented in Storm. We do not use the topological, warm start, or MEC-collapsing optimizations. We consider three orderings:

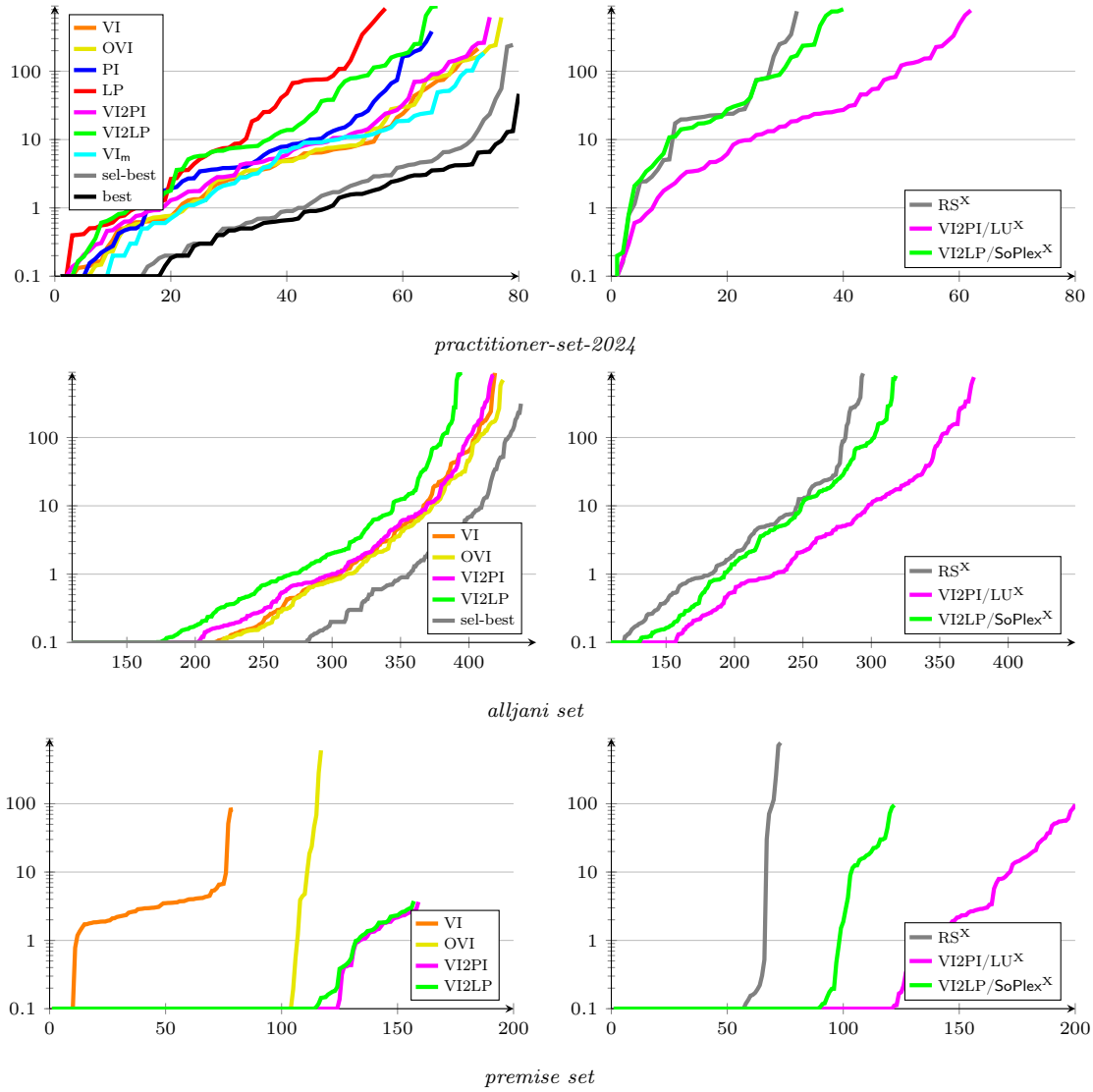


Fig. 16: Comparison of MDP model checking algorithms on different benchmark sets, considering floating point algorithms on the left and exact algorithms on the right.

– breadth-first search back-to-front (bfs-b2f): The default ordering in both Storm and mcsta explores the state space in a breadth-first search and then evaluates states backwards, intuitively starting with the states "closest to the end".
– breadth-first search front-to-back (bfs-f2b): The default ordering reversed, i.e. starting with the initial state.
– random (rnd): A random permutation.

The state ordering affects all three classes of algorithm. For **VI**, the default ordering bfs-b2f is clearly optimal, as visible in the quantile plot as well as the scatter plots. Using a random ordering makes the solving 2-8 times slower (with the scatter plot for bfs-b2f similar, but omitted for space). For **LP**, the effect is least pronounced. The bfs-b2f ordering is preferable, but outperformed by the other orderings on some benchmark instances. For **PI**, the effect is most extreme: the random state-ordering solves 9 fewer instances than bfs-b2f and is also consistently worse. Interestingly, the bfs-f2b ordering appears preferable for PI.

6.4 Influence of Hardware

Common practice in research papers on probabilistic model checking is to fix a hardware platform and execute the benchmarks on this hardware platform. Claims about the algorithm and its implementation are then generalized beyond this hardware. In recent years, it has become increasingly popular to put the environments for

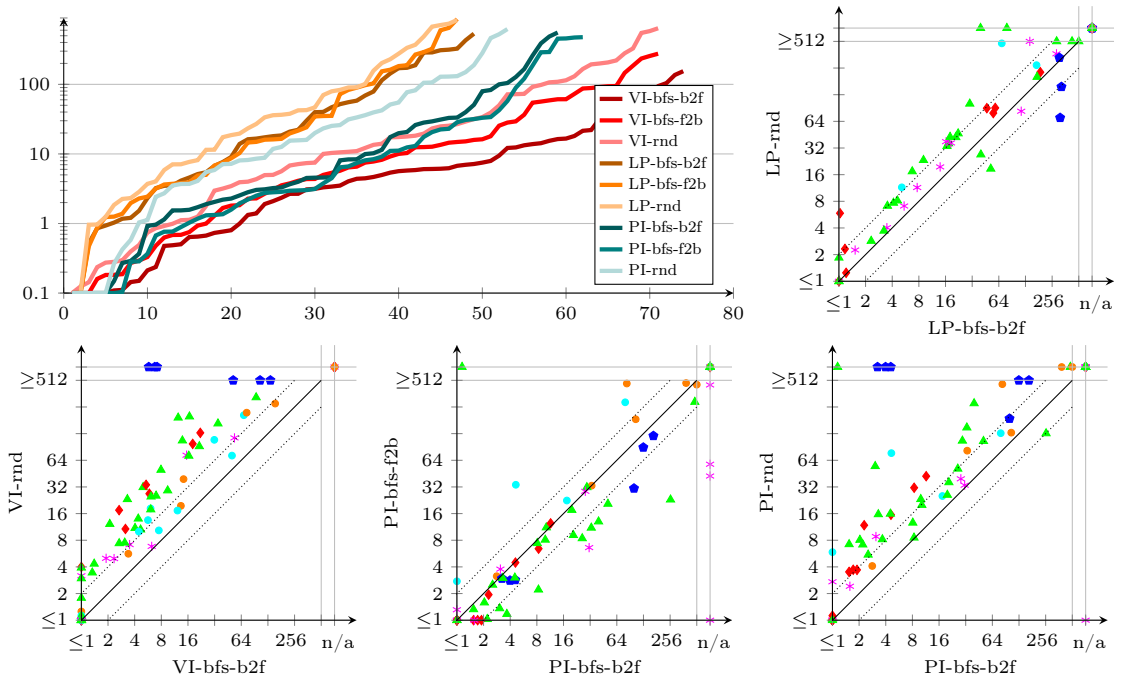


Fig. 17: Impact of permuting the states

these experiments into a Docker container for easier reproduction of the benchmarks. This led us to formulate the following questions:

HW-Q1 What is the difference between running our experiments within a Docker container and on bare metal?

HW-Q2 What is the difference in running our experiments on different hardware?

The goal is *not* to assess different hardware or to understand which hardware is most suitable for probabilistic model checking. Instead, we aim to understand whether the common practice to benchmark on a single type of host machine potentially invalidates claims about the algorithms and their implementations.

Setup. We used 9 machines, listed in Table 6. These machines are between 1 and 9 years old with a nice spread over Intel and AMD processors. For ARM64-based machines, we encountered some challenges discussed below and thus only report preliminary results for this platform. We can run our experiments either on the operating system directly, or within a Docker container, outlined below. A platform combines a machine with the choice of host. As benchmark set, we use the *practitioner-set-2024*. On each platform, we executed six configurations: VI_s, OVI_s, VI_m, OVI_m, VI2LP_s, and VI2PI_s. We chose them as reasonable and relevant subset of all configurations. As timeout, we use 10 minutes. Empirically, our experiments run between 8-14 hours on our platforms.

Results in a nutshell. The general trends hold when running our benchmarks on the different hardware and on either bare metal or inside a container. In particular, the quantile plots qualitatively *all* agree. However, a closer look highlights that some benchmarks perform significantly differently on different hardware platforms. It is not uncommon that for over ten benchmarks, two algorithms have a 30% performance difference on one platform and (almost) no performance difference on a different platform. We therefore recommend that authors reporting results on probabilistic model checking use different hardware to validate their claims, unless the differences between algorithms are orders of magnitude apart or the benchmark set is sufficiently large (at least as large as *practitioner-set-2024*).

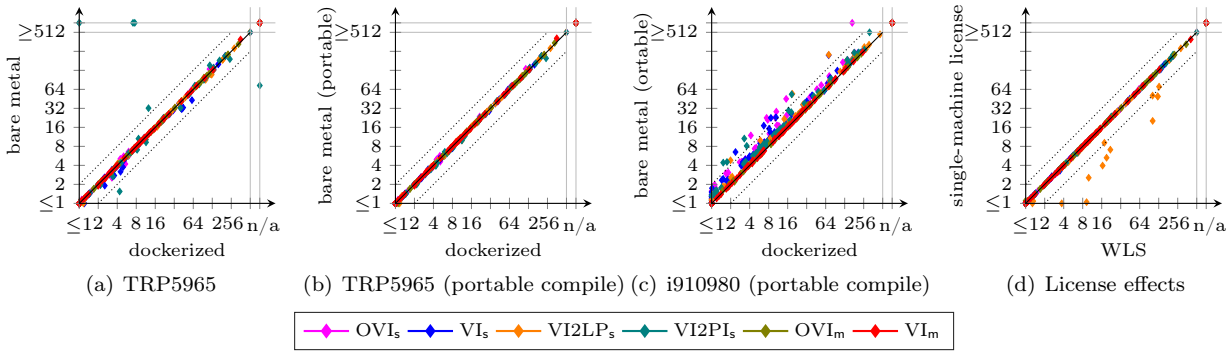
6.4.1 Using Docker Containers

To answer to **HW-Q1**, we run our Docker container and (independently) also install all tools directly in the OS on two machines: The (AMD) TRP5965 with a GCC 12 compiler and the (Intel) i910980 with a GCC 11 compiler.

Our Docker container. The Docker container is based on a Ubuntu 24.04 LTS standard container for the x86 platform with a GCC 13 compiler. The container is extended by the Storm Dockerfile (which includes the dependencies and Storm). We further extend this container with

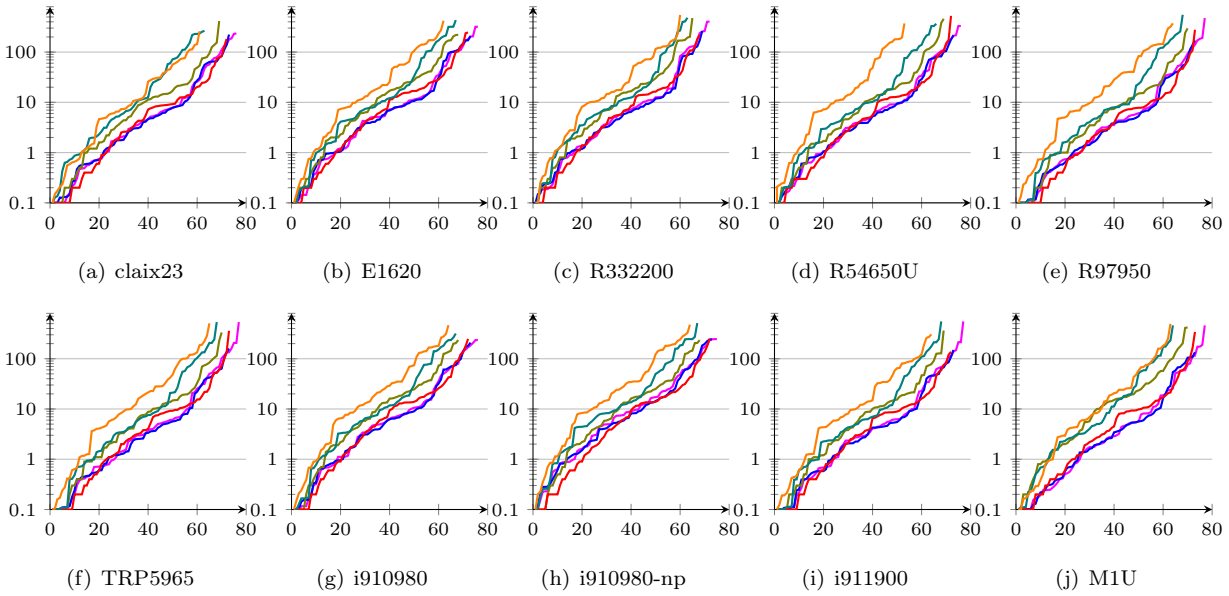
Table 6: Hardware platforms used in Section 6.4.

Identifier	CPU	RAM	OS	Comment
claix23	Intel Xeon 8468 Sapphire	unclear	Rocky Linux 8.9	HPC with slurm
E1620	Intel Xeon E5-1620 v4	2x16GB DDR4 2400	Ubuntu 20.04	9 years old
i911900	Intel Core i9-11900K	4x32GB DDR4 2400	Ubuntu 22.04	
i910980	Intel Core i9-10980XE	2x32GB DDR4 3200	Ubuntu 22.04	
M1U	Apple M1 Ultra	128GB LPDDR5-6400 (unified)	Mac OS 14.2	MacStudio
R32200	AMD Ryzen 3 2200G	2x8GB DDR4 3200	Win 10 Home	Docker gets only 10GB
R54650U	AMD Ryzen 5 PRO 4650U	1x16GB DDR4 3200	Debian 13	Laptop
R97950	AMD Ryzen 9 7950X3D	4x32GB DDR5 4800	Ubuntu 22.04	
TRP5965	AMD ThreadRipper Pro 5965	8x64GB DDR4 3200 (registered)	Debian 12	



(a) TRP5965 (b) TRP5965 (portable compile) (c) i910980 (portable compile) (d) License effects

Fig. 18: Performance on bare metal and inside a Docker container.



(a) claix23 (b) E1620 (c) R332200 (d) R54650U (e) R97950

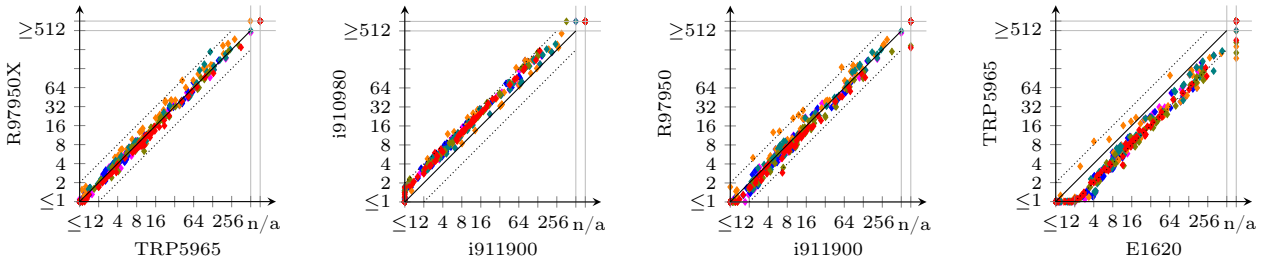
(f) TRP5965 (g) i910980 (h) i910980-np (i) i911900 (j) M1U

Fig. 19: Quantile plots for different hardware configurations (legend see Fig. 18)

the binary for mcsta and with our benchmark scripts[11]. When running the container, we mount an *academic WLS license* for Gurobi. With this setup, we make three observations:

1) Docker environments itself do not incur a tangible runtime overhead. In Fig. 18, we show the performance effects of running inside a Docker container. If a Docker container would yield a constant runtime penalty, then we would expect all points to lie on a perfect line parallel to the main diagonal. Figure 18(a) shows installing and running all tools as default on the TRP5965. Notably, the points for mcsta, which runs with the same precompiled binaries embedding a .NET runtime environment everywhere, lie on a straight line. This observation also

[11] The Docker container is available online and the instructions are part of the artifact.



(a) AMD platforms (b) Intel platforms (c) Similar platforms (d) Contrasting platforms

Fig. 20: Detailed comparison of performance of docker container on different hardware.

highlights that the TRP5965 yields very deterministic timings. However, in particular the PI configuration yields some stark outliers. As we show below, these are *not* due to running a Docker container.

2) Instruction sets matter. On the TRP5965, this effect is *solely* due to the compiler flags used to compile Storm inside a Docker file where the compiler does not use the complete available instruction set to increase portability of the Docker container. In Fig. 18(b), we run a version of Storm compiled with this flag on bare-metal: We observe no significant differences between the Docker container on the TRP5965.

3) The compiler version may matter. We now consider the i910980. Again, the mcsta points lie on a straight line, also indicating deterministic timings. However, if we run the same portably-compiled Storm version on the i910980, we still observe significant differences: In all cases, it seems to be *faster* to run Storm inside a Docker container. Our current best guess is that this is caused by differences between the two compiler versions and the operating systems, although different compiler versions are also present on the TRP5965. The quantile plots for Figs. 19(g) and 19(h) reflect the differences, where Storm implementations for all algorithms perform slightly better inside the Docker container, but where the overall picture remains the same.

Licensing challenges. Running Gurobi within Docker requires a special license that needs an active Internet connection to communicate with Gurobi servers for every benchmark. We mostly observe a < 1 second delay which varies over time and over machines, but also some remarkable differences between wall-clock time and CPU time for the LP configurations. Indeed, these also appear to be caused by the type of Gurobi license: Within our university's network and on the bare metal, we can use a single-machine academic license that does not constantly connect to Gurobi servers, see Fig. 18(d). [12]

[12] Considering CPU time rather than wall-clock time is complicated if one measures timings for parts of the code.

Docker and ARM hosts. We note that the Docker container can be emulated using QEMU on an ARM64 platform, which may skew performance but also prevents executing mcsta because the .NET 8.0 runtime it uses is not supported under QEMU emulation at the time of writing (due to using features/instructions that are currently not properly emulated, leading to mcsta crashing).

6.4.2 Using Different Hardware

To answer **HW-Q2**, we primarily run our Docker container on 7 different platforms with an x86-64 host. We create quantile plots for 10 different platforms. In addition to the performance inside the Docker container on 7 different machines, Fig. 19(a) reflects a bare-metal run on the claix23 cluster used for the main experiments, Fig. 19(h) runs bare-metal on the i910980 and was mentioned in the section above, and Fig. 19(j) runs bare-metal on the M1U. We make the following two observations:

1) Similar trends on all platforms. The quantile plots in Fig. 19 look similar, although close inspection shows differences in the number of benchmarks solved by PI and LP, in particular, and more subtly, the relative performance of VI on Storm and mcsta, especially for solving roughly 60 benchmarks. In Fig. 21 on page 31, where we fix a method and compare different hardware platforms, we see that for the four VI configs, the shapes of the curves look very similar. For PI and LP, there is more spread. This spread for LP-based methods *may* actually be an effect of the use of WLS licenses for Gurobi as discussed above. It is notable that compared with the hardware tested here, running benchmarks on claix appears to be favourable for LP-based approaches and unfavourable for PI-based approaches.

2) Individual differences are significant. In Fig. 20, we compare four pairs of machines, while Fig. 22 on page 31 gives a richer set of such comparisons. Comparing either two AMD (Fig. 20(a)) or two Intel (Fig. 20(b))

platforms running the same container shows a spread of up to 40% in performance differences and ranging over different configurations. The difference between two comparably-powered machines, one AMD and one Intel as in Fig. 20(c), is generally a bit higher and more configuration-dependent. If one additionally compares the machines that differ significantly in their computational power, these differences gets larger, see Fig. 20(d). We have not investigated whether some machines predictably perform better, e.g., on MEC collapsing, or on large state spaces (where a larger cache may be beneficial).

ARMv8/64 platforms. Due to the increased popularity of the ARMv8 architecture, we executed some preliminary experiments on an M1 chip. While Fig. 19(j) shows a familiar image, we need to be more careful. We executed the benchmark set twice and observed that the timings on the M1U are not as deterministic, see Fig. 23(a) on page 31. The additional quantile plot, however, looks similar. It is also interesting to see that while the quantile plot looks similar to the x86-64 platforms, the individual spread compared to comparable Intel (Fig. 23(c)) and AMD (Fig. 23(d)) platforms is significant.

6.5 Discussion and Takeaways

The empirical study above is necessarily a snapshot. Some results may be unsurprising, e.g. that the benchmark set really matters, while other results are highly specialised and bound to change, e.g. that the performance of specific hardware or tools is lacking. Beyond these concrete statements, we believe that our empirical evaluation uncovers three take-away messages, each addressing a different perspective on model checking tools.

If the goal is to model-check a fixed MDP and model checking that MDP is a bottleneck, then one should investigate using a different configuration instead of the default of one's favourite tool. In particular, Fig. 16 shows that the virtual best may perform significantly better than that configuration. To find a good configuration, it often suffices to test a limited set of four configurations, outlined in Section 6.2. Finally, it may be worth investigating the performance using smaller variants of the given MDP, but it is an open question how to correctly define such a smaller but representative MDP.

If the goal is to analyse the next MDP model checking algorithm then it is advisable to refrain from making statements based on small numbers of benchmarks or non-diverse benchmark sets. For example, the *qvbs* benchmark set was not structurally diverse, in particular not containing benchmarks with MECs. Statements about handling MECs based on the *qvbs* set were premature and our experiments show that also some tool implementations to handle MECs may be sub-optimal. When considering (subsets of) small benchmark sets, hardware effects may explain some of the differences—see Section 6.4. More generally, it is essential to continuously push for a more diverse benchmark set. Beyond the benchmark set, the use of preprocessing steps is heavily benchmark-dependent and it may be misleading to enable a particular preprocessing step across different approaches as shown in Section 6.1.4.

If a model checker appears as a subroutine then great care is advised. Consider the case where the subroutine is called on an MDP that is obtained by some transformation. The performance gains of a different transformation to a (standard) model checking routine on a (standard) MDP may be due to some hidden structure and may not indicate a better translation, but merely that this translation is bad for the default model checking routine. We note that such effects may even depend on simple state reorderings which can significantly change the performance of the methods (see Section 6.3).

7 Conclusion

This paper evaluates MDP model checking from a practical perspective, shining a spotlight on the main methods to analyse a given MDP for undiscounted, indefinite-horizon properties. While the literature has a traditional focus on value iteration-based methods, we also review approaches based on policy iteration and linear programming in depth. We highlight a range of possible variations to the approaches, and illustrate how several approaches may yield incorrect results. The larger part of the paper discusses an extensive empirical evaluation along multiple axes: It shows that overall, value iteration remains the fastest algorithm, even when compared to LP-based approaches that in theory run in polynomial time. This general trend does not help when model checking individual MDPs, however: For them, either PI or LP-based approaches may be faster. Our evaluation debunks some myths about the role of preprocessing steps and presents take-away messages that raise awareness for the large influence that seemingly innocent choices have on the evaluation of a particular algorithm.

These take-away messages are a call for future work: Indeed, our paper shows that independent tools with strong reference implementations of the different approaches are necessary to properly evaluate the merits of an algorithm, but also that users of such tools must

receive help in choosing a set of meaningful defaults. Therefore, we must understand the key characteristics of MDPs that can be the basis for a prognosis of algorithm performance on a particular MDP. While we *observed* the behaviour of the different algorithms and have some intuition into what makes certain structures (or e.g. the *premise* set in particular) hard, an entire research question of its own is to properly identify and quantify these structural properties. As a first step, we provide the *practitioner-set-2024*. It strikes the balance between being structurally diverse and being small enough so that many tool configurations can be evaluated on it. Of course, further investigation is required to find out which structures are most relevant, in order to be able to improve the *practitioner-set-2024*, making it more representative while maintaining the balance.

References

1. Allamigeon, X., Dadush, D., Loho, G., Natura, B., Végh, L.A.: Interior point methods are not worse than simplex. In: FOCS, pp. 267–277. IEEE (2022)
2. Anand, R., Aggarwal, D., Kumar, V.: A comparative analysis of optimization solvers. Journal of Statistics and Management Systems **20**(4), 623–635 (2017). DOI 10.1080/09720510.2017.1395182
3. Ashok, P., Chatterjee, K., Daca, P., Kretínský, J., Meggendorfer, T.: Value iteration for long-run average reward in Markov decision processes. In: CAV (1), *LNCS*, vol. 10426, pp. 201–221. Springer (2017). DOI 10.1007/978-3-319-63387-9_10
4. Auger, D., de Montjoye, X.B., Strozecki, Y.: A generic strategy improvement method for simple stochastic games. In: MFCS, *LIPIcs*, vol. 202, pp. 12:1–12:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). DOI 10.4230/LIPICS.MFCS.2021.12
5. Azeem, M., Evangelidis, A., Kretínský, J., Slivinskiy, A., Weininger, M.: Optimistic and topological value iteration for simple stochastic games. CoRR **abs/2207.14417** (2022). DOI 10.48550/arXiv.2207.14417
6. Azeem, M., Evangelidis, A., Kretínský, J., Slivinskiy, A., Weininger, M.: Optimistic and topological value iteration for simple stochastic games. In: ATVA, *Lecture Notes in Computer Science*, vol. 13505, pp. 285–302. Springer (2022). DOI 10.1007/978-3-031-19992-9_18
7. Baier, C., de Alfaro, L., Forejt, V., Kwiatkowska, M.: Model checking probabilistic systems. In: Handbook of Model Checking, pp. 963–999. Springer (2018)
8. Baier, C., D'Argenio, P.R., Größer, M.: Partial order reduction for probabilistic branching time. In: A. Cerone, H. Wiklicky (eds.) 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL), *Electronic Notes in Theoretical Computer Science*, vol. 153-2, pp. 97–116. Elsevier (2005). DOI 10.1016/J.ENTCS.2005.10.034
9. Baier, C., Größer, M., Ciesinski, F.: Partial order reduction for probabilistic systems. In: 1st International Conference on Quantitative Evaluation of Systems (QEST), pp. 230–239. IEEE Computer Society (2004). DOI 10.1109/QEST.2004.1348037
10. Baier, C., Hermanns, H., Katoen, J.P.: The 10,000 facets of MDP model checking. In: Computing and Software Science, *LNCS*, vol. 10000, pp. 420–451. Springer (2019). DOI 10.1007/978-3-319-91908-9_21
11. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008). URL https://mitpress.mit.edu/books/principles-model-checking
12. Baier, C., Klein, J., Klüppelholz, S., Märcker, S.: Computing conditional probabilities in Markovian models efficiently. In: TACAS, *LNCS*, vol. 8413, pp. 515–530. Springer (2014). DOI 10.1007/978-3-642-54862-8_43
13. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: CAV (1), *LNCS*, vol. 10426, pp. 160–180. Springer (2017). DOI 10.1007/978-3-319-63387-9_8
14. Baier, C., Klein, J., Leuschner, L., Parker, D., Wunderlich, S.: Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In: CAV (1), *Lecture Notes in Computer Science*, vol. 10426, pp. 160–180. Springer (2017). DOI 10.1007/978-3-319-63387-9_8
15. Balaji, N., Kiefer, S., Novotný, P., Pérez, G.A., Shirmohammadi, M.: On the complexity of value iteration. In: ICALP, *LIPIcs*, vol. 132, pp. 102:1–102:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). DOI 10.4230/LIPIcs.ICALP.2019.102
16. Bellman, R.: A Markovian decision process. Journal of Mathematics and Mechanics **6**(5), 679–684 (1957)
17. Berkelaar, M., Eikland, K., Notebaert, P.: Introduction to lp_solve 5.5.2.11. URL https://lpsolve.sourceforge.net/5.5/. Accessed 2023-01-25.
18. Bertsekas, D.P., Tsitsiklis, J.N.: An analysis of stochastic shortest path problems. Math. Oper. Res. **16**(3), 580–595 (1991). DOI 10.1287/moor.16.3.580
19. Boyd, S.P., Vandenberghe, L.: Convex Optimization. Cambridge University Press (2014)
20. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Kretínský, J., Kwiatkowska, M.Z., Parker, D., Ujma, M.: Verification of Markov decision processes using learning algorithms. In: ATVA, *LNCS*, vol. 8837, pp. 98–114. Springer (2014). DOI 10.1007/978-3-319-11936-6_8
21. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: Quantitative model and tool interaction. In: TACAS (2), *Lecture Notes in Computer Science*, vol. 10206, pp. 151–168 (2017). DOI 10.1007/978-3-662-54580-5_9
22. Budde, C.E., Hartmanns, A., Klauck, M., Kretínský, J., Parker, D., Quatmann, T., Turrini, A., Zhang, Z.: On correctness, precision, and performance in quantitative verification – QComp 2020 competition report. In: ISoLA (4), *LNCS*, vol. 12479, pp. 216–241. Springer (2020). DOI 10.1007/978-3-030-83723-5_15
23. Chatterjee, K., Henzinger, T.A.: Value iteration. In: 25 Years of Model Checking, *LNCS*, vol. 5000, pp. 107–138. Springer (2008). DOI 10.1007/978-3-540-69850-0_7
24. Chen, T., Forejt, V., Kwiatkowska, M.Z., Parker, D., Simaitis, A.: Automatic verification of competitive stochastic systems. Formal Methods Syst. Des. **43**(1), 61–92 (2013). DOI 10.1007/S10703-013-0183-7
25. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer (2018). DOI 10.1007/978-3-319-10575-8
26. Cubuktepe, M., Jansen, N., Junges, S., Katoen, J., Topcu, U.: Convex optimization for parameter synthesis in mdps. IEEE Trans. Autom. Control. **67**(12), 6333–6348 (2022). DOI 10.1109/TAC.2021.3133265. URL https://doi.org/10.1109/TAC.2021.3133265
27. Dai, P., Mausam, Weld, D.S., Goldsmith, J.: Topological value iteration algorithms. J. Artif. Intell. Res. **42**, 181–209 (2011). URL https://www.jair.org/index.php/jair/article/view/10725

28. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reduction and refinement strategies for probabilistic analysis. In: PAPM-PROBMIV, *Lecture Notes in Computer Science*, vol. 2399, pp. 57–76. Springer (2002). DOI 10.1007/3-540-45605-8_5
29. D'Argenio, P.R., Niebert, P.: Partial order reduction on concurrent probabilistic programs. In: 1st International Conference on Quantitative Evaluation of Systems (QEST), pp. 240–249. IEEE Computer Society (2004). DOI 10.1109/QEST.2004.1348038
30. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: CAV, *LNCS*, vol. 4144, pp. 81–94. Springer (2006)
31. Eisentraut, J., Kelmendi, E., Kretínský, J., Weininger, M.: Value iteration for simple stochastic games: Stopping criterion and learning algorithm. Inf. Comput. **285**(Part), 104886 (2022). DOI 10.1016/j.ic.2022.104886
32. Fearnley, J.: Exponential lower bounds for policy iteration. In: ICALP (2), *LNCS*, vol. 6199, pp. 551–562. Springer (2010). DOI 10.1007/978-3-642-14162-1_46
33. Feng, L., Kwiatkowska, M.Z., Parker, D.: Automated learning of probabilistic assumptions for compositional reasoning. In: FASE, *Lecture Notes in Computer Science*, vol. 6603, pp. 2–17. Springer (2011). DOI 10.1007/978-3-642-19811-3_2
34. Forejt, V., Kwiatkowska, M.Z., Norman, G., Parker, D.: Automated verification techniques for probabilistic systems. In: SFM, *Lecture Notes in Computer Science*, vol. 6659, pp. 53–113. Springer (2011). DOI 10.1007/978-3-642-21455-4_3
35. Forejt, V., Kwiatkowska, M.Z., Parker, D.: Pareto curves for probabilistic model checking. In: ATVA, *LNCS*, vol. 7561, pp. 317–332. Springer (2012). DOI 10.1007/978-3-642-33386-6_25
36. Free Software Foundation: The GNU Multiple Precision Arithmetic Library. URL https://gmplib.org/. Accessed 2023-01-25.
37. Funke, F., Jantsch, S., Baier, C.: Farkas certificates and minimal witnesses for probabilistic reachability constraints. In: TACAS (1), *LNCS*, vol. 12078, pp. 324–345. Springer (2020)
38. Ge, D., Huangfu, Q., Wang, Z., Wu, J., Ye, Y.: Cardinal Optimizer (COPT) user guide (2022). URL https://guide.coap.online/copt/en-doc
39. GetFEM project: Gmm++ Library. URL https://getfem.org/gmm/. Accessed 2023-01-25.
40. Giro, S.: Optimal schedulers vs optimal bases: An approach for efficient exact solving of Markov decision processes. Theor. Comput. Sci. **538**, 70–83 (2014). DOI 10.1016/j.tcs.2013.08.020
41. Giro, S., D'Argenio, P.R., Fioriti, L.M.F.: Partial order reduction for probabilistic systems: A revision for distributed schedulers. In: M. Bravetti, G. Zavattaro (eds.) 20th International Conference on Concurrency Theory (CONCUR), *Lecture Notes in Computer Science*, vol. 5710, pp. 338–353. Springer (2009). DOI 10.1007/978-3-642-04081-8_23
42. Gleixner, A.M., Steffy, D.E., Wolter, K.: Improving the accuracy of linear programming solvers with iterative refinement. In: ISSAC, pp. 187–194. ACM (2012)
43. Gleixner, A.M., Steffy, D.E., Wolter, K.: Iterative refinement for linear programming. Tech. Rep. 3, ZIB, Takustr. 7, 14195 Berlin (2016). DOI 10.1287/ijoc.2016.0692
44. GNU Project: GLPK (GNU Linear Programming Kit). URL http://www.gnu.org/software/glpk/glpk.html
45. Google: Glop – linear optimization. URL https://developers.google.com/optimization/lp. Accessed 2023-01-25.
46. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010). URL http://eigen.tuxfamily.org
47. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2022). URL https://www.gurobi.com
48. Haddad, S., Monmege, B.: Reachability in MDPs: Refining convergence of value iteration. In: RP, *LNCS*, vol. 8762, pp. 125–137. Springer (2014)
49. Haddad, S., Monmege, B.: Interval iteration algorithm for MDPs and IMDPs. Theor. Comput. Sci. **735**, 111–131 (2018). DOI 10.1016/j.tcs.2016.12.003
50. Hall, J., Galabova, I., Gottwald, L., Feldmeier, M.: HiGHS – high performance software for linear optimization. URL https://www.maths.ed.ac.uk/hall/HiGHS/. Accessed 2023-01-25.
51. Hartmanns, A.: Correct probabilistic model checking with floating-point arithmetic. In: TACAS (2), *LNCS*, vol. 13244, pp. 41–59. Springer (2022). DOI 10.1007/978-3-030-99527-0_3
52. Hartmanns, A., Hermanns, H.: The Modest Toolset: An integrated environment for quantitative modelling and verification. In: TACAS, *LNCS*, vol. 8413, pp. 593–598. Springer (2014). DOI 10.1007/978-3-642-54862-8_51
53. Hartmanns, A., Junges, S., Quatmann, T., Weininger, M.: A practitioner's guide to MDP model checking algorithms. In: TACAS (1), *Lecture Notes in Computer Science*, vol. 13993, pp. 469–488. Springer (2023). DOI 10.1007/978-3-031-30823-9_24
54. Hartmanns, A., Kaminski, B.L.: Optimistic value iteration. In: CAV (2), *LNCS*, vol. 12225, pp. 488–511. Springer (2020). DOI 10.1007/978-3-030-53291-8_26
55. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: TACAS, *LNCS*, vol. 11427, pp. 344–350. Springer (2019). DOI 10.1007/978-3-030-17462-0_20
56. Hensel, C.: The probabilistic model checker storm: symbolic methods for probabilistic model checking. Ph.D. thesis, RWTH Aachen University, Germany (2018). URL http://publications.rwth-aachen.de/record/752011
57. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **24**(4), 589–610 (2022). DOI 10.1007/s10009-021-00633-z
58. Huangfu, Q., Hall, J.A.J.: Parallelizing the dual revised simplex method. Math. Program. Comput. **10**(1), 119–142 (2018). DOI 10.1007/s12532-017-0130-5
59. IBM: IBM ILOG CPLEX Optimizer. URL https://www.ibm.com/analytics/cplex-optimizer. Accessed 2023-01-25.
60. Junges, S., Jansen, N., Seshia, S.A.: Enforcing almost-sure reachability in pomdps. In: CAV (2), *Lecture Notes in Computer Science*, vol. 12760, pp. 602–625. Springer (2021)
61. Junges, S., Torfah, H., Seshia, S.A.: Runtime monitors for Markov decision processes. In: CAV (2), *Lecture Notes in Computer Science*, vol. 12760, pp. 553–576. Springer (2021). DOI 10.1007/978-3-030-81688-9_26
62. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Formal Methods Syst. Des. **36**(3), 246–280 (2010). DOI 10.1007/s10703-010-0097-6
63. Kretínský, J., Meggendorfer, T.: Of cores: A partial-exploration framework for Markov decision processes. Log. Methods Comput. Sci. **16**(4) (2020). URL https://lmcs.episciences.org/6833
64. Kretinsky, J., Ramneantu, E., Slivinskiy, A., Weininger, M.: Comparison of algorithms for simple stochastic games. Inf. Comput. (2022). DOI 10.1016/j.ic.2022.104885
65. Kumar, A., Zilberstein, S.: History-based controller design and optimization for partially observable MDPs. In: ICAPS, vol. 25, pp. 156–164 (2015)
66. Kwiatkowska, M.Z., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods Syst. Des. **29**(1), 33–78 (2006). DOI 10.1007/s10703-006-0005-2

67. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. **94**(1), 1–28 (1991). DOI 10.1016/0890-5401(91)90030-6
68. Littman, M.L., Dean, T.L., Kaelbling, L.P.: On the complexity of solving Markov decision problems. In: UAI, pp. 394–402. Morgan Kaufmann (1995)
69. Mathur, U., Bauer, M.S., Chadha, R., Sistla, A.P., Viswanathan, M.: Exact quantitative probabilistic model checking through rational search. Formal Methods Syst. Des. **56**(1), 90–126 (2020)
70. McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: ICML, *ACM International Conference Proceeding Series*, vol. 119, pp. 569–576. ACM (2005). DOI 10.1145/1102351.1102423
71. Meggendorfer, T.: PET - A partial exploration tool for probabilistic verification. In: ATVA, *Lecture Notes in Computer Science*, vol. 13505, pp. 320–326. Springer (2022). DOI 10.1007/978-3-031-19992-9_20
72. Meggendorfer, T., Weininger, M.: Playing games with your PET: extending the partial exploration tool to stochastic games. In: CAV, Lecture Notes in Computer Science. Springer (2024 (to appear))
73. MOSEK ApS: The MOSEK Optimization Suite 10.0.34. URL https://docs.mosek.com/latest/intro/index.html. Accessed 2023-01-25.
74. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS, *LNCS*, vol. 4963, pp. 337–340. Springer (2008). DOI 10.1007/978-3-540-78800-3_24
75. Phalakarn, K., Takisaka, T., Haas, T., Hasuo, I.: Widest paths and global propagation in bounded value iteration for stochastic games. In: CAV (2), *LNCS*, vol. 12225, pp. 349–371. Springer (2020). URL https://doi.org/10.1007/978-3-030-53291-8_19
76. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics. Wiley (1994). DOI 10.1002/9780470316887
77. Quatmann, T., Katoen, J.P.: Sound value iteration. In: CAV (1), *LNCS*, vol. 10981, pp. 643–661. Springer (2018). DOI 10.1007/978-3-319-96145-3_37
78. Roux, S.L., Pérez, G.A.: The complexity of graph-based reductions for reachability in Markov decision processes. In: FoSSaCS, *Lecture Notes in Computer Science*, vol. 10803, pp. 367–383. Springer (2018). DOI 10.1007/978-3-319-89366-2_20
79. Wimmer, R., Jansen, N., Vorpahl, A., Ábrahám, E., Katoen, J.P., Becker, B.: High-level counterexamples for probabilistic automata. Log. Methods Comput. Sci. **11**(1) (2015)
80. Wimmer, R., Kortus, A., Herbstritt, M., Becker, B.: Probabilistic model checking and reliability of results. In: DDECS, pp. 207–212. IEEE Computer Society (2008). DOI 10.1109/DDECS.2008.4538787
81. Ye, Y.: The simplex and policy-iteration methods are strongly polynomial for the Markov decision problem with a fixed discount rate. Mathematics of Operations Research **36**(4), 593–603 (2011)

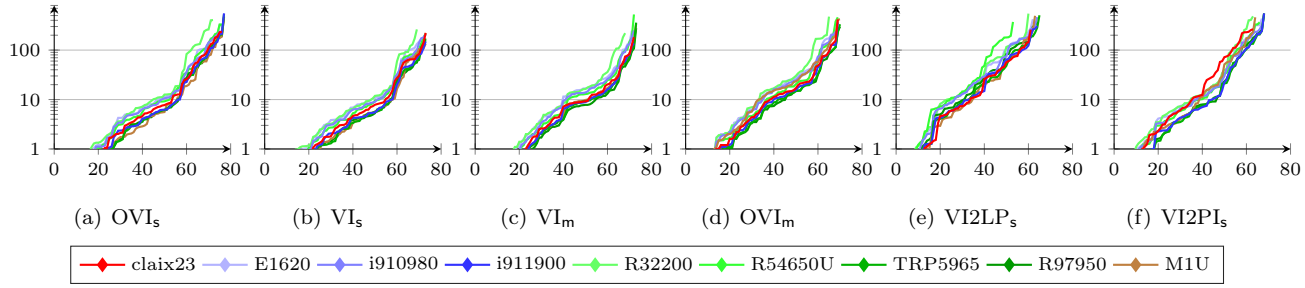


Fig. 21: Quantile plots with fixed tool configurations on various hardware.

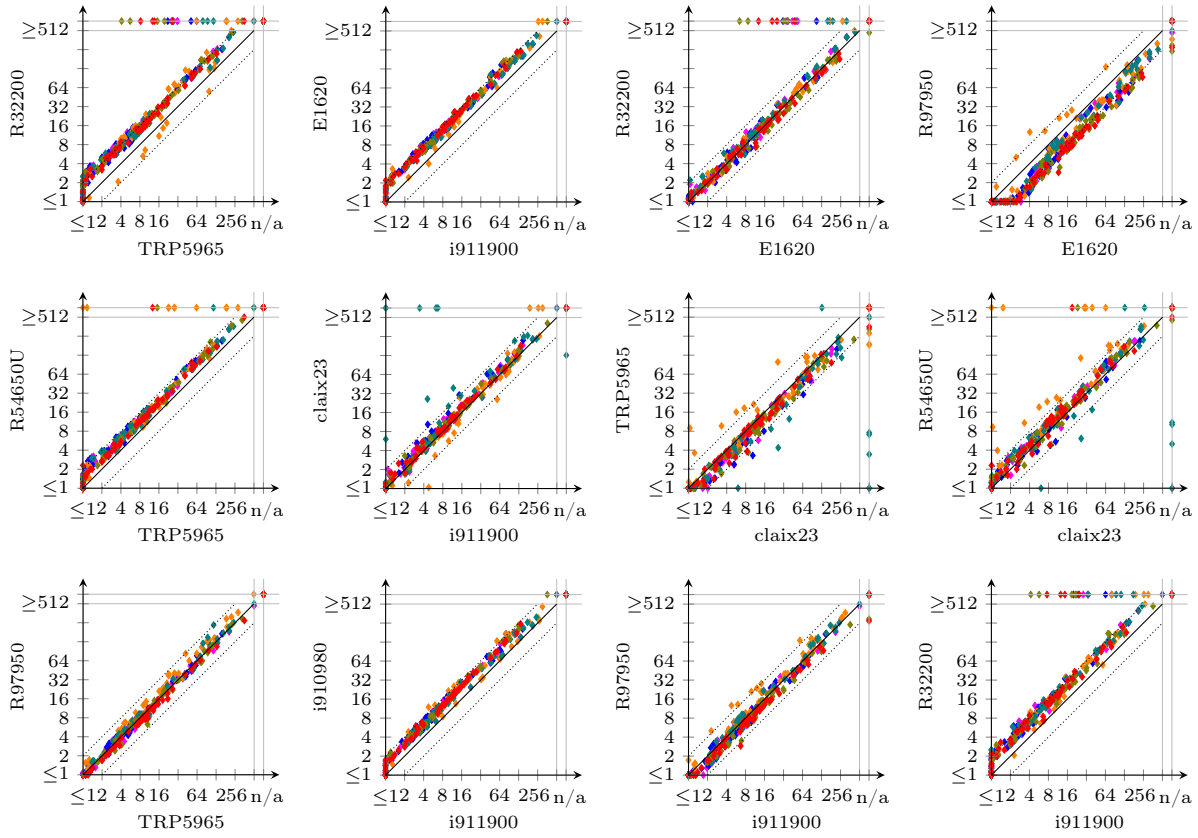


Fig. 22: Detailed comparison of different hardware. A column is analogous to a single plot in Fig. 20.

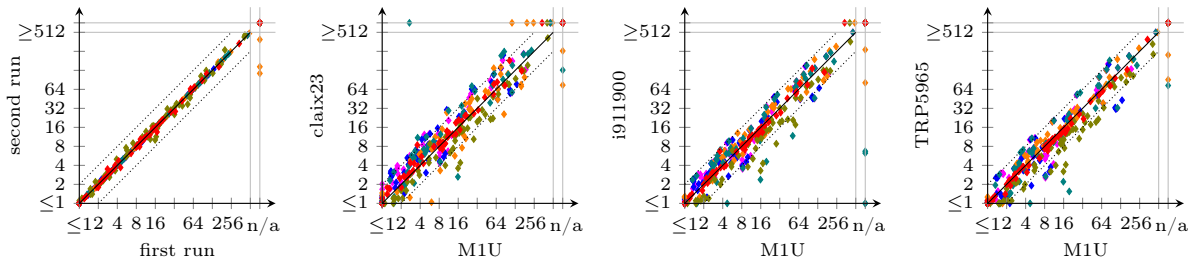


Fig. 23: Preliminary comparison to an ARM64 platform.