Information and Computation 289 (2022) 104885

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Comparison of algorithms for simple stochastic games ☆

Jan Křetínský, Emanuel Ramneantu, Alexander Slivinskiy, Maximilian Weininger *

Technical University of Munich, Germany

ARTICLE INFO

Article history:
Received 4 November 2020
Received in revised form 16 September 2021
Accepted 23 February 2022
Available online 1 March 2022

Keywords:
Formal methods
Probabilistic verification
Stochastic games
Algorithms
Value iteration
Strategy iteration
Quadratic programming

ABSTRACT

Simple stochastic games are turn-based 2½-player zero-sum graph games with a reachability objective. The problem is to compute the winning probabilities as well as the optimal strategies of both players. In this paper, we compare the three known classes of algorithms – value iteration, strategy iteration and quadratic programming – both theoretically and practically. Further, we suggest several improvements for all algorithms, including the first approach based on quadratic programming that avoids transforming the stochastic game to a stopping one. Our extensive experiments show that these improvements can lead to significant speed-ups. We implemented all algorithms in PRISM-games 3.0, thereby providing the first implementation of quadratic programming for solving simple stochastic games.

1. Introduction

Simple stochastic games (SGs), e.g. [18], are zero-sum games played on a graph by players Maximizer and Minimizer, who choose actions in their respective vertices (also called states). Each action is associated with a probability distribution determining the next state to move to. The objective of Maximizer is to maximize the probability of reaching a given target state; the objective of Minimizer is the opposite.

The basic decision problem is to determine whether Maximizer can ensure a reachability probability above a certain threshold if both players play optimally. This problem is among the rare and intriguing combinatorial problems that are in **NP** ∩ **co-NP** [19], but whether it belongs to **P** is a major and long-standing open problem. Further, several other important problems can be reduced to SG, for instance parity games, mean-payoff games, discounted-payoff games and their stochastic extensions [8].

Besides the theoretical interest, SGs are a standard model in control and verification of stochastic reactive systems [23, 11]; see e.g. [44] for an overview over various recent case studies. Further, since Markov decision processes (MDP) [40] are

☆ This research was funded in part by the German Research Foundation (DFG) projects 383882557 *Statistical Unbounded Verification (SUV)* and 427755713 *Group-By Objectives in Probabilistic Verification (GOPro)*. It is the extended version of the paper of the same name that appeared at GandALF 20 [37]. On the theoretical side, we added the full technical proofs and more description of quadratic programming. For the experiments, we extended the description of the experimental setup, included the complete tables for the runtime comparison and added an analysis of memory consumption. Moreover, we investigate the behaviour of quadratic programming on models with many actions per state.

* Corresponding author.
E-mail address: maximilian.weininger@tum.de (M. Weininger).

https://doi.org/10.1016/j.ic.2022.104885

a special case with only one player, SGs can serve as abstractions of large MDPs [31] or provide robust versions of MDPs when precise transition probabilities are not known [14,45].

There are three classes of algorithms for computing the reachability probability in simple stochastic games, as surveyed in [19]: value iteration (VI), strategy iteration (SI, also known as policy iteration) and quadratic programming (QP). In [19], they all required the SG to be transformed into a certain normal form, among other properties ensuring that the game is stopping. This not only blows up the size of the SG, but also changes the reachability probability; however, it is possible to infer the original probability. For VI and SI, this requirement has since been lifted, e.g. [10,7], but not for QP.

While searching for a polynomial algorithm, there were several papers on VI and SI; however, the theoretical improvements so far are limited to subexponential variants [38,21] and variants that are fast for SG with few random vertices [24,30], i.e. games where most actions yield a successor deterministically. QP was not looked at, as it is considered common knowledge that it performs badly in practice.

There exist several tools for solving games: GAVS+ [17] offers VI and SI for SGs, among other things. However, it is more for educational purposes than large case studies and currently not maintained. GIST [12] performs qualitative analysis of stochastic games with ω-regular objectives. For MDPs (games with a single player), we refer to [26] for an overview of existing tools. Most importantly, PRISM-games 3.0 [36] is a recent tool that offers algorithms for several classes of games. However, for SGs with a reachability objective, it offers only variants of value iteration, none of which give a guarantee, so the result might be arbitrarily far off. Thus, currently no tool offers a precise solution method for large simple stochastic games.

Our contribution is the following:

- We provide an extension of the quadratic programming approach that does not require the transformation of the SG into a stopping SG in normal form.
- We propose several optimizations for the algorithms, inspired by [34], including an extension of topological value iteration from MDPs [22,4].
- We implement VI with guarantees on precision as well as SI, QP and our optimizations in PRISM-games 3.0—thereby providing the first implementations of QP for SGs—and experimentally compare them on both the realistic case studies of PRISM-games and on interesting handcrafted corner cases.

Related work

We sketch the recent developments of each class of algorithms and then several further directions.

Value iteration is a standard solution method also for MDPs [40]. For a long time, the stopping criterion of VI was such that it could return arbitrarily imprecise results [25]. In principle, the computation has to run for an exponential number of steps in order to be able to give a precise answer [10]. However, recently several heuristics that give guarantees and are typically fast were proposed for MDPs [6,25,41,28], as well as for SGs [32,39]. A learning-based variant of VI for MDPs [6] was also extended to SGs [32]. The variant of VI from [30] requires the game to be in normal form, blowing up its size, and is good only if there are few random vertices, as it depends on the factorial of this number; it is impractical for almost all case studies considered in this paper.

Strategy iteration was introduced in [29]. A randomized version was given in [19], and subexponential versions in [38,21]. Another variant of SI was proposed in [43], however there is no evidence that it runs in polynomial time. The idea of considering games with few random vertices (as in the already mentioned works of [21,30]) was first introduced in [24], where they exhaustively search a subspace of strategies, which is called f-strategies.

To the best of our knowledge, quadratic programming as solution method for simple stochastic games was not investigated further after the first mention in [19]. However, convex QPs are solvable in polynomial time [33]. If it was possible to encode the problem in a convex QP of polynomial size, this would result in a polynomial algorithm. The encoding we provide in this paper can however be exponential in the size of the game.

Further related works consider other variants of the model, for example concurrent stochastic games, see e.g. [27] for the complexity of SI and VI and [9] for strategy complexity, or games with limited information [2]. Furthermore, one can consider other objectives, e.g. ω-regular objectives [11], total reward [4], mean payoff [46] or combinations of objectives [16, 1].

2. Preliminaries

We now introduce the model of stochastic games, define the semantics by the standard means of infinite paths and strategies and then define the important concept of end components, which are subgraphs of stochastic games that are problematic for all three classes of algorithms.

A *probability distribution* on a finite set X is a mapping $\delta : X \to [0, 1]$, such that $\sum_{x \in X} \delta(x) = 1$. The set of all probability distributions on X is denoted by $\mathcal{D}(X)$.

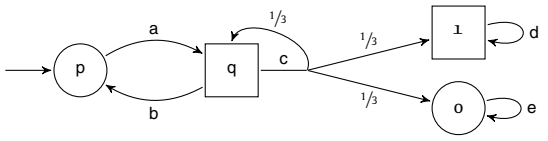

Fig. 1. An example of an SG with $S = \{\mathsf{p}, \mathsf{q}, \imath, \mathsf{o}\}$, $S_{\square} = \{\mathsf{q}, \imath\}$, $S_{\bigcirc} = \{\mathsf{p}, \mathsf{o}\}$, the initial state p and the set of actions $\mathsf{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$; $\mathsf{Av}(\mathsf{p}) = \{\mathsf{a}\}$ with $\delta(\mathsf{p}, \mathsf{a})(\mathsf{q}) = 1$; $\mathsf{Av}(\mathsf{q}) = \{\mathsf{b}, \mathsf{c}\}$ with $\delta(\mathsf{q}, \mathsf{b})(\mathsf{p}) = 1$ and $\delta(\mathsf{q}, \mathsf{c})(\mathsf{q}) = \delta(\mathsf{q}, \mathsf{c})(\imath) = \delta(\mathsf{q}, \mathsf{c})(\mathsf{o}) = \frac{1}{3}$. For actions with only one successor, we do not depict the transition probability 1.

2.1. Stochastic games

Now we define stochastic games, in literature often referred to as simple stochastic games or turn-based stochastic two-player games with a reachability objective. As opposed to the notation of e.g. [18], we do not have special stochastic nodes, but rather a probabilistic transition function. See Fig. 1 for an example of an SG.

Definition 1 (SG). A *stochastic game (SG)* is a tuple $(S, S_{\square}, S_{\bigcirc}, \mathsf{s}_0, \mathsf{A}, \mathsf{Av}, \delta, F)$ where

- S is a finite set of *states* partitioned[1] into the sets $S_{\square}$ and $S_{\bigcirc}$ of states of the player *Maximizer* and *Minimizer*[2] respectively
- $\mathsf{s}_0 \in S$ is the *initial* state
- A is a finite set of *actions*
- $\mathsf{Av} : S \to 2^{\mathsf{A}}$ assigns to every state a set of *available* actions
- $\delta : S \times \mathsf{A} \to \mathcal{D}(S)$ is a *transition function* that given a state s and an action $\mathsf{a} \in \mathsf{Av}(\mathsf{s})$ yields a probability distribution over *successor* states. We slightly abuse notation and write $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}')$ instead of $\delta(\mathsf{s}, \mathsf{a})(\mathsf{s}')$.
- $F \subseteq S$ is a set of target states. Without loss of generality we can assume that every target state only has one action that is a self-loop with probability 1, because we consider the reachability objective.

A *Markov decision process (MDP)* is a special case of SG where $S_{\bigcirc} = \emptyset$, and a Markov chain (MC) is a special case of an MDP, where for all $\mathsf{s} \in S : \left|\mathsf{Av}(\mathsf{s})\right| = 1$.

Without loss of generality we assume that SGs are non-blocking, so for all states s we have $\mathsf{Av}(\mathsf{s}) \neq \emptyset$. For a state s and an available action $\mathsf{a} \in \mathsf{Av}(\mathsf{s})$, we denote the set of successors by $\mathsf{Post}(\mathsf{s}, \mathsf{a}) := \{\mathsf{s}' \mid \delta(\mathsf{s}, \mathsf{a}, \mathsf{s}') > 0\}$. Finally, for any set of states $T \subseteq S$, we use $T_{\square}$ and $T_{\bigcirc}$ to denote the states in T that belong to Maximizer and Minimizer, whose states are drawn in the figures as squares $\square$ and circles $\bigcirc$, respectively.

2.2. Semantics: paths, strategies and values

The semantics of SGs is given in the usual way by means of strategies, the induced Markov chain and the respective probability space, as follows: An *infinite path* ρ is an infinite sequence $\rho = \mathsf{s}_0\mathsf{a}_0\mathsf{s}_1\mathsf{a}_1 \cdots \in (S \times \mathsf{A})^{\omega}$, such that for every $i \in \mathbb{N}$ we have $\mathsf{a}_i \in \mathsf{Av}(\mathsf{s}_i)$ and $\mathsf{s}_{i+1} \in \mathsf{Post}(\mathsf{s}_i, \mathsf{a}_i)$. *Finite paths* are defined analogously as elements of $(S \times \mathsf{A})^* \times S$.

As this paper deals with the probabilistic reachability objective, we can restrict our attention to memoryless deterministic strategies, which are optimal for this objective [18]. A *strategy* of Maximizer, respectively Minimizer, is a function $\sigma : S_{\square} \to \mathsf{A}$, respectively $S_{\bigcirc} \to \mathsf{A}$, such that $\sigma(\mathsf{s}) \in \mathsf{Av}(\mathsf{s})$ for all s. A pair (σ, τ) of memoryless deterministic strategies of Maximizer and Minimizer induces a Markov chain $\mathsf{G}^{\sigma,\tau}$, as for every state the strategies select a single action. The Markov chain induces a unique probability distribution $\mathbb{P}_{\mathsf{s}}^{\sigma,\tau}$ over measurable sets of infinite paths [3, Ch. 10].

We write $\Diamond F := \{\rho \mid \rho = \mathsf{s}_0\mathsf{a}_0\mathsf{s}_1\mathsf{a}_1 \cdots \in (S \times \mathsf{A})^{\omega} \wedge \exists i \in \mathbb{N}.\ \mathsf{s}_i \in F\}$ to denote the (measurable) set of all paths which eventually reach F. For each $\mathsf{s} \in S$, we define the *value* in s as

$$\mathsf{V}(\mathsf{s}) := \sup_{\sigma} \inf_{\tau} \mathbb{P}_{\mathsf{s}}^{\sigma,\tau}(\Diamond F) = \inf_{\tau} \sup_{\sigma} \mathbb{P}_{s}^{\sigma,\tau}(\Diamond F),$$

where the equality follows from [18]. Note that SGs with a reachability objective are *determined*, i.e. knowing the strategy of the opponent does not increase the chances of winning, provided the opponent plays optimally. Moreover, since there are only finitely many memoryless deterministic strategies, one could replace sup and inf with max and min.

[1] I.e., $S_{\square} \subseteq S$, $S_{\bigcirc} \subseteq S$, $S_{\square} \cup S_{\bigcirc} = S$, and $S_{\square} \cap S_{\bigcirc} = \emptyset$.

[2] The names are chosen, because Maximizer maximizes the probability of reaching the given target states, and Minimizer minimizes it.

The value is the least fixpoint of the so called *Bellman equations* [19]:

$$\mathsf{V}(s) = \begin{cases} 1 & \text{if } s \in F \\ \max_{a \in \mathsf{Av}(s)} \mathsf{V}(s,a) & \text{if } s \in S_{\square} \setminus F \\ \min_{a \in \mathsf{Av}(s)} \mathsf{V}(s,a) & \text{if } s \in S_{\bigcirc} \setminus F, \end{cases} \tag{1}$$

$$\text{with } \mathsf{V}(s,a) := \sum_{s' \in S} \delta(s,a,s') \cdot \mathsf{V}(s') \tag{2}$$

The states with no path to the target, so called sinks, are of special interest, as they have a value of 0. Moreover, they can be computed a priori by standard graph analysis. For example, the sinks are all states that are not visited by a backwards depth-first search beginning at the target states F. We denote the set of sinks as Z.

We are interested not only in the values $\mathsf{V}(s)$ for all $s \in S$, but also their ε-approximation, i.e. an approximation $\mathsf{L} : S \to \mathbb{Q}$ with $\left|\mathsf{V}(s) - \mathsf{L}(s)\right| < \varepsilon$. Moreover, we want to provide the corresponding (ε-)optimal strategies for both players. A strategy σ of Maximizer is (ε-)optimal, if for every Minimizer strategy τ we have $\mathbb{P}_s^{\sigma,\tau}(\Diamond F) \geq \mathsf{V}(s)$ (respectively $\geq \mathsf{V}(s) - \varepsilon$) and an optimal Minimizer strategy is defined analogously. Note that it suffices to have either the values or the optimal strategies, because from one we can infer the other as follows: given a pair of optimal strategies (σ, τ), the values can be computed by solving the induced Markov chain $\mathsf{G}^{\sigma,\tau}$. Given a vector of values for all states, an optimal pair of strategies can be computed by randomizing over all locally optimal actions in each state (e.g. $\sigma(s)$ randomizes over the set $\arg\max_{a \in \mathsf{Av}(s)} \mathsf{V}(s,a)$ for Maximizer states, and dually with $\arg\min$ for Minimizer). To get a deterministic strategy, we cannot only pick some locally optimal action, but additionally we have to ensure that Maximizer is not stuck in some cycle when playing the actions. One way of doing this is by looking at end components, which are the topic of the next subsection.

2.3. End components

When computing the values of states in an SG, we need to take special care of *end components* (EC). Intuitively, an EC is a subset of states of the SG, where the game can remain forever; i.e. given certain strategies of both players, there is no positive probability to exit the EC to some other state. ECs correspond to bottom strongly connected components of the Markov chains induced by some pair of strategies.

Definition 2 *(EC).* A non-empty set $T \subseteq S$ of states is an *end component (EC)* if there exists a non-empty set $B \subseteq \bigcup_{s \in T} \mathsf{Av}(s)$ of actions[3] such that

1. for each $s \in T, a \in B \cap \mathsf{Av}(s)$ we have $\mathsf{Post}(s,a) \subseteq T$,
2. for each $s, s' \in T$ there is a finite path $w = s a_0 \dots a_n s' \in (T \times B)^* \times T$, i.e. the path stays inside T and only uses actions in B.

An end component T is a *maximal end component (MEC)* if there is no other end component T' such that $T \subseteq T'$.

Given an SG G, the set of its MECs is denoted by $\mathsf{MEC}(\mathsf{G})$ and can be computed in polynomial time [20]. ECs are of special interest, because in ECs there can be multiple fixpoints of the Bellman equations (see Equation (1) and (2)).

Example 1. Consider the SG of Fig. 1. The set of states $T = \{p, q\}$ is an EC, as when playing only actions from $B = \{a, b\}$ the play remains in T forever. It is even a MEC, as there is no superset of T with this property.

We now show that setting the value of both states in T to any x with $\frac{1}{2} \leq x \leq 1$ is a fixpoint of the Bellman equations. Note that since $1 \in F$, $\mathsf{V}(1) = 1$, and since $o \in Z$, $\mathsf{V}(o) = 0$. State p only has action a that goes with probability 1 to q. So, since $\mathsf{V}(q) = x$, we also have $\mathsf{V}(p) = x$. Similarly, $\mathsf{V}(q,b) = x$. Finally, observe that $\mathsf{V}(q,c) = \frac{1}{3} + \frac{1}{3} \cdot x$, which is equal to x for $x = \frac{1}{2}$ and less than x for all $x > \frac{1}{2}$. Thus, for every $x \geq \frac{1}{2}$, it will be optimal for the Maximizer state q to use action b and we have $\mathsf{V}(q) = x$.

So we see that there are infinitely many fixpoints which are greater than the value, which is the least fixpoint of the Bellman equations, i.e. $\mathsf{V}(p) = \mathsf{V}(q) = \frac{1}{2}$. $\triangle$

As there can be multiple greater fixpoints of the Bellman equations, methods relying on iterative over-approximation of the value may not converge, as they can be stuck at some greater fixpoint than the value (cf. [32, Lemma 1] and Example 2 in the next section). This is why the original description of the algorithms [19] considered only stopping games. Stopping

[3] Note that this assumes that action names are unique. This can always be achieved by renaming actions, e.g. prepending every action with the state it is played from.

games are SGs where the set $F \cup Z$ is reached with probability 1, or equivalently games without ECs in the set $S \setminus (F \cup Z)$. The algorithms are theoretically applicable to arbitrary SGs, as for every non-stopping SG one can construct a stopping SG and infer the original value from solving the stopping SG. See Section 3.3.2 for a description of this approach and Section 4.1.4 for a discussion of the practical drawbacks.

3. State of the art algorithms

In this section, we describe the existing algorithms for solving simple stochastic games, namely value iteration, strategy iteration and quadratic programming (Section 3.1, 3.2 and 3.3 respectively). The input for all of them is an SG $\mathsf{G} = (S, S_{\Box}, S_{\bigcirc}, \mathsf{s}_0, \mathsf{A}, \mathsf{Av}, \delta, F)$. Value iteration additionally needs a precision ε, given as a rational number. After termination, all of them return a vector of values for each state (ε-precise for BVI) and the corresponding (ε-)optimal strategies. Quadratic programming in its current form only works on stopping SGs in a certain normal form.

3.1. Bounded value iteration

Value Iteration (VI), see e.g. [40], is the most common algorithm for solving MDPs and SGs, and the only method implemented in PRISM-games [36]. Originally, VI only computed a convergent sequence of under-approximations; however, as it was unclear how to stop, results returned by model checkers could be off by arbitrary amounts [25]. Thus, it was extended to also compute a convergent over-approximation [32]. The resulting algorithm is called *bounded value iteration (BVI)*.

The basic idea of BVI is to start from a vector L_0 respectively U_0 that definitely is an under-/over-approximation of the value, i.e. for every state $\mathsf{L}_0(\mathsf{s}) \leq \mathsf{V}(\mathsf{s}) \leq \mathsf{U}_0(\mathsf{s})$. Then the algorithm repeatedly applies so called *Bellman updates*, i.e. it uses a version of Equation (1) as follows (the equation for the over-approximation is obtained by replacing L with U):

$$\mathsf{L}_n(\mathsf{s}) = \begin{cases} \max_{\mathsf{a} \in \mathsf{Av}(\mathsf{s})} \mathsf{L}_{n-1}(\mathsf{s},\mathsf{a}) & \text{if } \mathsf{s} \in S_{\Box} \\ \min_{\mathsf{a} \in \mathsf{Av}(\mathsf{s})} \mathsf{L}_{n-1}(\mathsf{s},\mathsf{a}) & \text{if } \mathsf{s} \in S_{\bigcirc} \end{cases}, \tag{3}$$

where $\mathsf{L}_{n-1}(\mathsf{s},\mathsf{a})$ is computed from $\mathsf{L}_{n-1}(\mathsf{s})$ as in Equation (2). Since V is the least fixpoint of the Bellman equations, $\lim_{n\to\infty} \mathsf{L}_n$ converges to V. However, the over-approximation U need not converge in the presence of ECs.

Example 2. Consider the SG of Fig. 1 with the EC $\{\mathsf{p},\mathsf{q}\}$. Let $\mathsf{U}_0 = 1$ for p, q and $\imath$ and $\mathsf{U}_0 = 0$ for o. Then we have $\mathsf{U}_0(\mathsf{q},\mathsf{b}) = 1$ and $\mathsf{U}_0(\mathsf{q},\mathsf{c}) = 2/3$. Thus, q will pick action b, as it promises a higher value, and the over-approximation does not change. This happens, because looking at the current upper bound, Maximizer is under the impression that staying in the EC yields a higher value. However, it actually reduces the probability to reach the target to 0. So the algorithm has to perform an additional step to inform states in an EC that they should not depend on each other, but on the best exit. In this case, $\mathsf{U}_1(\mathsf{q}) = \mathsf{U}_0(\mathsf{q},\mathsf{c}) = 2/3$. △

In fact, it does not suffice to only decrease the upper bound of ECs, but a more in-depth graph analysis is required to detect the problematic subsets of states, so called *simple end components* (SEC) [32, Definition 5]. These SECs cannot be found a priori, because they depend on the values of their exits. Thus, candidate SECs are guessed according to the current lower bound. When the lower bound eventually converges to the value, the true SECs are found. The over-approximation of states in the SECs has to be decreased ("deflated" in the words of [32]) to the over-approximation of the best exit from the SEC. Formally, the best exit is defined as

$$\mathsf{best\_exit}_{\mathsf{U}}(T) = \max_{\substack{\mathsf{s} \in T_{\Box} \\ \neg \mathsf{Post}(\mathsf{s},\mathsf{a}) \subseteq T}} \mathsf{U}(\mathsf{s},\mathsf{a}),$$

where U is the over-approximation in the current iteration. Decreasing the approximation for the SEC candidates to the best exit always results in a correct over-approximation [32, Lemma 3] and suffices to converge to the true value in the limit [32, Theorem 2]. Algorithm 1 shows the full BVI algorithm from [32].

There also is a simulation based asynchronous version of BVI (see [32, Section 4.4]) which can perform very well on models with a certain structure [35]. It updates states encountered by simulations, and guides those simulations to explore only the relevant part of the state space. If only few states are relevant for convergence, this algorithm is fast; if large parts of the state space are relevant or there are many cycles in the game graph that slow the simulations down, this adaption of BVI performs badly.

3.2. Strategy iteration

In contrast to value iteration, the approach of *strategy iteration (SI)* [29] does not compute a sequence of value-vectors, but instead a sequence of strategies. Starting from an arbitrary strategy of Maximizer, we repeatedly compute the best response of Minimizer and then greedily improve Maximizer's strategy. The resulting sequence of Maximizer strategies is monotonic and converges to the optimal strategy [7, Theorem 3]. The pseudocode for strategy iteration is given in Algorithm 2.

Algorithm 1 Bounded value iteration algorithm from [32].

```
1: procedure BVI(precision ε > 0)
2:     for s ∈ S do      # Initialization
3:         L(s) = 0      # Lower bound
4:         U(s) = 1      # Upper bound
5:     for s ∈ F do L(s) = 1      # Value of targets is determined a priori
6:     repeat
7:         L, U get updated according to Eq. (3)      # Bellman updates
8:         for every SEC candidate T do
9:             for s ∈ T do
10:                U(s) ← best_exit_U(T)      # Decrease U to best exit
11:    until U(s) − L(s) < ε for all s ∈ S      # Guaranteed error bound
```

Algorithm 2 Strategy iteration.

```
1: procedure SI
2:     σ′ ← arbitrary Maximizer attractor strategy      # Proper initial strategy
3:     repeat
4:         σ ← σ′
5:         for s ∈ S do
6:             L(s) ← inf_τ ℙ_s^{σ,τ}(◇F)      # Solve MDP for opponent
7:         for s ∈ S_□ do
8:             if σ(s) ∈ arg max_{a∈Av(s)} L(s, a) then
9:                 σ′(s) ← σ(s) # Only change σ on strict improvement
10:            else
11:                σ′(s) ← any element of arg max_{a∈Av(s)} L(s, a)
12:    until σ = σ′
```

Note that in non-stopping SGs (games with ECs) the initial Maximizer strategy cannot be completely arbitrary, but it has to be *proper*, i.e. ensure that either a target or a sink state is reached almost surely; it must not stay in some EC, as otherwise the algorithm might not converge to the optimum due to problems similar to those described in Example 2. In Algorithm 2, we use the construction of the *attractor strategy* [7, Section 5.3] to ensure that our initial guess is a proper strategy (Line 2). It first analyses the game graph to find the sink states Z. Then, it performs a backwards breadth first search, starting from the set of both target and sink states. A state discovered in the i-th iteration of the search has to choose some action that reaches a state discovered in the (i-1)-th iteration with positive probability. Such a state exists by construction, and the choice ensures that the initial strategy reaches the set of target or sink states almost surely.

When a Maximizer strategy is fixed, the main loop of Algorithm 2 solves the induced MDP G^σ (Line 6). We discuss the different ways to do this in Section 4.2. The algorithm need not remember the Minimizer strategy, but only uses the computed value estimates L to greedily update Maximizer's strategy (Line 11); note that here $\mathsf{L}(s,a)$ is again computed from $\mathsf{L}(s)$ as in Equation (2). For termination, it is necessary that we only update the strategy if switching strictly improves the value (see Line 8), as otherwise we might cycle infinitely between equivalent strategies. The algorithm stops when the Maximizer strategy does not change any more in one iteration. We can then compute the values and the corresponding Minimizer strategy by solving the induced MDP G^σ.

3.3. Quadratic programming

As mentioned before, the approach of Quadratic programming currently requires that the input SG is in a certain normal form. We first recall how the approach works for a game in that normal form and then describe how to transform an arbitrary SG into normal form.

3.3.1. The quadratic program

Quadratic programming (QP) [19] works by encoding the graph of the SG in a system of constraints. The only global (and local) optimum of the objective function under these constraints is 0, and it is attained if and only if the variable for every state is set to the value of that state [19, Section 3.1]. The proof as well as the construction of the quadratic program relies on the game being in a certain normal form, which consists of four conditions:

- **2Act**: For all $s \in S : |\mathsf{Av}(s)| \leq 2$.
- **No1Act**: If $|\mathsf{Av}(s)| = 1$, then s is a target or a sink.
- **½Probs**: For all $s, s' \in S, a \in \mathsf{Av}(s) : \delta(s,a,s') \in \{0, 0.5, 1\}$.
- **Stopping game**: There are no ECs in G (except for the sinks and targets).

We shortly discuss the advantage of each condition of the normal form: the reason for **2Act** and **No1Act** is that the objective function of the QP requires every (non-sink and non-target) state to have exactly 2 successors. Arguing about a game with

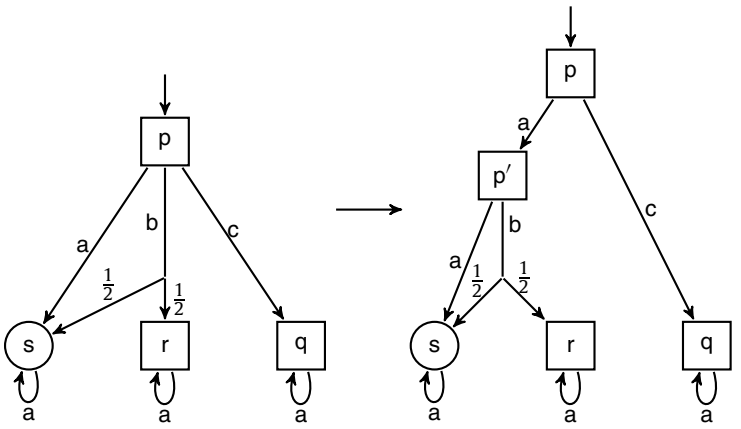

Fig. 2. An example of transforming an SG into one fulfilling the **2Act**-constraint. p has more than two actions, so a binary subtree is built up where every state except for the leaves s, r and q has exactly two actions.

average nodes instead of actions mapping to arbitrary probability distribution simplified the proofs, which is the advantage of **½Probs**. **Stopping game** was necessary, because of the problem of non-convergence in end components, as in Example 2. We describe the procedure from [18] to transfer an arbitrary SG into a polynomially larger SG in normal form in the next Section 3.3.2.

We now state the quadratic program for an SG in normal form as given in [19], but adjusted to our notation. Intuitively, the objective function is 0 if all summands are 0. And the summand for some state s is 0 if its value $V(s)$ is equal to the value of one of its actions $V(s,a)$ or $V(s,b)$. The program is quadratic, since all states (except targets and sinks) have exactly two successors and hence the summand for every state is a quadratic term. The constraints encode the game, ensuring that Maximizer/Minimizer states use the action with the highest/lowest value and fixing the values of targets and sinks. Note the additional definition of $V(s,a)$, which assumes that all occurring non-trivial probabilities are ½.

$$
\begin{aligned}
\text{minimize} \quad & \sum_{\substack{s \in S \\ Av(s)=\{a,b\}}} (V(s) - V(s,a))(V(s) - V(s,b)) \\
\text{subject to} \quad & V(s) \geq V(s,a) && \forall s \in S_{\square} : |Av(s)| = 2, \forall a \in Av(s) \\
& V(s) \leq V(s,a) && \forall s \in S_{\bigcirc} : |Av(s)| = 2, \forall a \in Av(s) \\
& V(s) = 1 && \forall s \in F \\
& V(s) = 0 && \forall s \in Z
\end{aligned}
$$

$$
\text{where } V(s,a) = \left\{ \begin{array}{ll} V(s') & \text{for } Post(s,a) = \{s'\} \\ \frac{1}{2}V(s') + \frac{1}{2}V(s'') & \text{for } Post(s,a) = \{s', s''\} \end{array} \right\}
$$

3.3.2. Transforming an arbitrary stochastic game into normal form

We describe the constructions of [18] to transform an arbitrary SG into one in normal form that is only polynomially larger. For each of the conditions that are required for normal form, i.e. for **2Act**, **No1Act**, **½Probs** and **Stopping game**, we give the respective transformation.

2Act: In normal form, every state s that is not an absorbing state must have at most two actions. To transform an arbitrary SG into one complying with **2Act**, take every state s with $|Av(s) > 2|$ and construct a binary tree as illustrated in Fig. 2. Two actions of s are taken and given to a new vertex v'. State s has then one action leading to the additional state v' instead of its previous two actions. This can be done iteratively until there is a binary tree where s is the root and has only two actions. Note that every other inner node in this tree also has two actions, and the leaves are exactly $v_1, v_2, ..., v_k$. If a state s has $n \geq 2$ actions, then after the transformation there are $n-2$ additional states. Every newly added state belongs to the same player as the original state s.**No1Act:** Every state that is not a target or sink must have more than one action. We can add a second action to every state that is missing one. This action leads to a target in the case of $s \in S_{\bigcirc}$ and otherwise to a sink. If there is no sink in S, we can introduce an artificial sink. In any optimal strategy, neither player would choose the additional action, as the other option is at least as good as the additional one. Therefore, the additional actions do not influence the value of any state.

In Fig. 3, s_0 has an additional action leading to o. Every state $s \in (S_{\bigcirc} \setminus \{o\})$ should have an action leading to 1, but we omit this to improve the readability of the figure.

½Probs: The normal form requires that transition probabilities are either 0, 0.5 or 1. This implies that every action has either one or two successors. Let $s \in S$ be a state which has an action a that has an arbitrary amount of successors $v_1, v_2, ..., v_k$

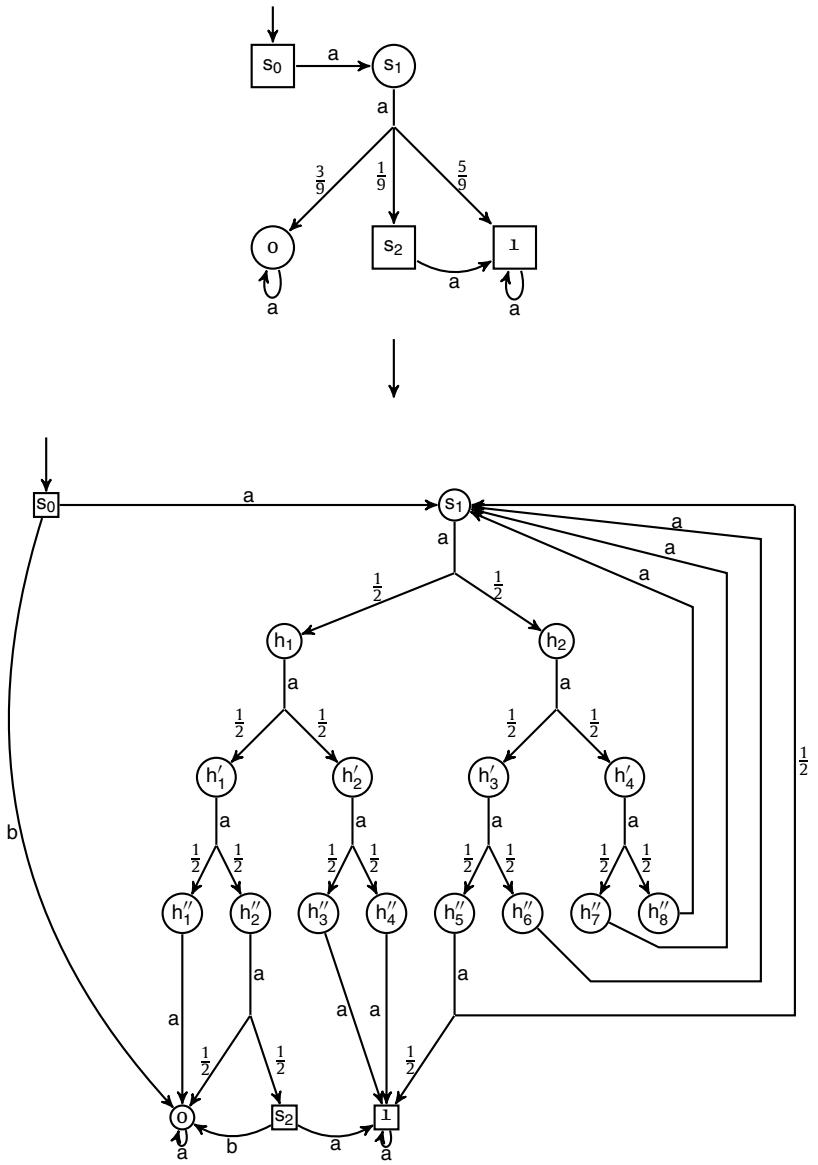

Fig. 3. An example of an arbitrary SG that gets transformed into Condon's normal form. States $h''_1, h''_2, ..., h''_8$, h'_1, h'_2, h'_3, h'_4 and h_1, h_2 are introduced to fulfill the **½Probs**-constraint. State s_2 has now a second action leading to sink o, as otherwise it would not comply with **No1Act**. All the ◯-states with only one action except the sink o must have an action leading to 1 but we omit these for the sake of readability.

with rational transitions probabilities $p_i := \delta(\mathsf{s}, \mathsf{a}\ v_i) \in [0,1] \subset \mathbb{Q}$. Consider the greatest common divisor q of all occurring transition probabilities of (s, a). Let q' be the smallest power of 2 such that $q' \geq q$, i.e. $q' = 2^k, k \in \mathbb{N} : 2^{k-1} < q < 2^k = q'$.

Create $1/2 \cdot q'$ new vertices, each with one action and two transitions with probability 0.5. Out of the q' many transitions, $p_i \cdot q$ lead to v_i. If a vertex has two transitions assigned that lead to the same state, the two transitions are unified to one with transition probability 1. The $q' - q$ remaining transitions lead to s. From the new vertices, we build up a binary tree with s as root such that s and every new state have one action each with two transitions that have only transition probabilities of 0, 0.5 or 1, and such that the probability of reaching v_i from s is p_i.

Fig. 3 illustrates an example. s_1 has an action a with transition probabilities $3/9, 1/9, 5/9$. The common divisor q of all these fractions is 9. The next power of 2 is 4, so we have $q = 9 \leq 16 = 2^4 = q'$. We need 8 states $h''_1, h''_2, ..., h''_8$ that store the 16 transitions. Every transition has a probability of $1/2$.

- The probability to reach o from s is $3/9$, therefore 3 transitions have to lead there. We let h''_1 lead to o with probability 1 and h''_2 with one of the two transitions.
- The probability to reach s_2 from s is $1/9$, therefore the second transition of h''_2 leads in s_2.
- The probability to reach 1 from s is $5/9$, therefore h''_3, h''_4 and one transition of h''_5 lead to 1.

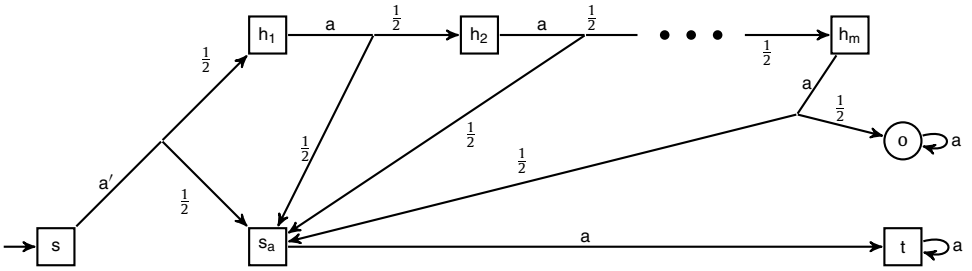

Fig. 4. An example of adding the ε-transitions to a simple SG where initially there were only the initial state s and a target state t and the action a with $\mathsf{Post}(s,a) = \mathsf{Post}(t,a) = t$. The action of s_a is simulating the outcome of taking action a in s in the original SG while $h_1, h_2, \ldots h_m$ add the ϵ needed if the initial SG would be non-stopping. For better readability, we omit the superscript s, a for all states h_i.

- The remaining transitions lead back to s.

To connect $h''_1, h''_2, \ldots, h''_8$ to s a binary tree is constructed with h'_1, h'_2, h'_3, h'_4 and h_1, h_2.

Stopping game: To avoid the possibility of never reaching any absorbing state, we add a transition with a small probability ε leading to a sink-state o to every action. If the players pick strategies that trap the play in a MEC in the original SG, the play would almost surely reach o in the modified SG. If the ε is chosen sufficiently small, one can infer the value in the original SG from the modified SG [18]. This is due to the fact that the value of an SG must be rational, where the denominator is at most $4^{|S|}$, and thus we can round the value in the modified SG to the nearest fraction with denominator $4^{|S|}$.

In Fig. 4, we illustrate the transformation. Although the initial SG is already stopping in this example, we chose this simple game for explanatory purposes. For the ε-transitions that have to be introduced to comply with the **Stopping game** constraint, [19] suggest something smarter instead of constructing a binary tree: for every state-action pair (s,a) add a new state s_a which has only one action that has the same successors and transition probabilities as (s,a) in the initial game. The state s instead has a new action a' with ½-probabilities of leading either to s_a or to the state $h_1^{s,a}$ of a chain of $m \in \mathbb{N}$ (sufficiently large) new states $h_1^{s,a}, h_2^{s,a}, \ldots h_m^{s,a}$. Each state $h_i^{s,a}$ with $i \in [m-1]$ has only one action with two ½-probabilities leading to either $h_{i+1}^{s,a}$ or s_a. The action of $h_m^{s,a}$ also has probabilities of ½ and leads either to a sink o or to s_a. Thus, playing a' in s now has a $\varepsilon = (1/2)^m$ chance of going to a sink, and with the remaining probability behaves as before.

4. Improvements

In this section, we first generalize QP to be applicable to arbitrary SGs, thereby omitting the costly transformations into the normal form. Then, we identify a hyperparameter of SI, namely that the way in which the MDP is solved after fixing one strategy can be varied. Finally, we provide two optimizations that are applicable to all three algorithms.

4.1. Quadratic programming for general stochastic games

Every transformation into the normal form (see Section 3.3.2) adds additional states or actions to the SG. We want to change the constraints of the QP so that it can deal with arbitrary SGs. This way, we avoid blowing up the SG, which is good since the time for solving the QP depends on the size of the SG.

4.1.1. 2Act

In order to drop the constraint that every state has at most two actions, we can generalize the summand of a state s in the objective function to $\prod_{a \in \mathsf{Av}(s)} (\mathsf{V}(s) - \mathsf{V}(s,a))$. The resulting program is no longer quadratic, as for a state with n actions now the objective function has order n. Thus, in the experiments, we report the verification times of both (i) a higher-order optimization problem for the original SG as well as (ii) a QP for the SG that was transformed to comply with **2Act**.

One more step is needed to ensure that the objective function still is correct: we have to duplicate one term $(\mathsf{V}(s) - \mathsf{V}(s,a))$ in Minimizer states with an odd number of actions. This ensures that the objective function cannot become negative. A more detailed explanation is given in the proof of the following lemma.

Lemma 1. *Consider a higher-order optimization problem for an SG* G *which minimizes the objective function* $\sum_{s \in S} \prod_{a \in \mathsf{Av}(s)} (\mathsf{V}(s) - \mathsf{V}(s,a))$*, where additionally for Minimizer states* s *with an odd number of actions, the term* $(\mathsf{V}(s) - \mathsf{V}(s,a))$ *is duplicated for an arbitrary action* $a \in \mathsf{Av}(s)$*. The constraints are as given*[4] *for the QP in Section 3.3.1, i.e. they encode the game* G*.*

Then the global optimum of the objective function is 0 and it is attained if and only if for every state $s \in S$ *and some action* $a \in \mathsf{Av}(s)$ *we have* $\mathsf{V}(s) = \mathsf{V}(s,a)$*.*

[4] Note that the constraint $\mathsf{V}(s) \sim \mathsf{V}(s,a)$ with $\sim \in \{\leq, \geq\}$ is specified for every action $a \in \mathsf{Av}(s)$ and now there can be more than two actions.

Proof. The objective function evaluates to 0 if and only if every summand is 0. This is the case if and only if at least one factor of every product is 0, which happens if and only if for every state $s \in S$ and some action $a \in Av(s)$ we have $V(s) = V(s,a)$. So if the objective function is 0, the solution vector V is a solution to the SG G.

It remains to show that 0 is indeed the global optimum of the higher-order optimization problem. For this, first observe that every summand is non-negative, i.e. greater or equal than 0: For every Maximizer state s, the constraint $V(s) \geq V(s,a)$ ensures that every factor of the product is non-negative, and thus the whole product is non-negative. For every Minimizer state, dually we know that every factor of the product is non-positive. Moreover, we have ensured that every product has an even number of factors. Thus, the product certainly is non-negative, since either all factors are smaller than 0 and the result is a positive number, or at least one factor is 0 and the result is 0. Finally, the argument that 0 can always be attained is analogous to the one in [19], since the number of actions per state does not affect the proof. □

4.1.2. No1Act

The **No1Act**-constraint compels every state $s \in S$ that is not a target or a sink to have more than one action. The transformation for complying with this constraint adds a second action to every state with only one action; however, the newly added action is chosen in such a way that it does not influence the value of the state, and can thus also be omitted.

Lemma 2. *Let $s \in S$ be a state in an SG with two actions a and b: a is the action s originally had, and b is the additional action inserted to comply with **No1Act**. Then it holds that $V(s) = V(s,a)$.*

Proof. We prove this by a case distinction over the player s belongs to. We only provide the proof for the case that $s \in S_{\square}$, as the other case is analogous. Let the strategy τ of the Minimizer be arbitrary. Let σ_b be a strategy for the Maximizer in which s takes action b. Per construction, b leads to a sink, and therefore it holds that $V_{\sigma_b,\tau}(s) = 0$. Let σ_a be a strategy for the Maximizer in which s takes action a. For every state $s' \in S$ it holds that $V(s') \geq 0$. Thus, $V_{\sigma_a,\tau}(s) \geq V_{\sigma_b,\tau} = 0$. It follows that every optimal Maximizer strategy σ may take action a in s. Since s allows a in every optimal strategy it holds that $V(s) = V(s,a)$. □

Thus, for a non-absorbing state s with only one action a, we can simplify the program by not including it in the objective function, but only adding a single constraint $V(s) = V(s,a)$.

4.1.3. ½Probs

To comply with the **½Probs**-requirement, every transition $\delta(s,a,s')$ with $s, s' \in S$ and $a \in Av(s)$ must have a probability of 0, 0.5 or 1. We can omit this constraint if we use the general definition of $V(s,a)$ as in Equation (2), summing the successors with their respective given transition probability.

Lemma 3. *Let G be an SG that does not comply with **½Probs**. The QP from Section 3.3.1 still is correct, i.e. returns the value vector of G, if we modify it such that $V(s,a) := \sum_{s' \in S} \delta(s,a,s') \cdot V(s')$.*

Proof. The proof, that the QP computes the value vector of the SG G from [19] does not rely on the fact the probabilities are all 0, 0.5 or 1. This restriction is only due to the different, more restrictive definition of SG. □

4.1.4. Stopping game

The final and most complicated constraint of the normal form is that we require the SG to be stopping. The transformation of an arbitrary game into a stopping one adds a transition to a sink with a small probability ε to every action, thus ensuring that a sink (or a target) is reached almost surely. This is problematic not only because the added transitions blow up the quadratic program, but even more so because of the following.

The ε-transitions modify the value of the game. Theoretically, this is no problem, because the value is rational and we know the greatest possible denominator q it can have. Thus, by choosing ε sufficiently small, we ensure that the modified value does not differ by more than $1/q$ and we can obtain the original value by rounding. However, practically, this denominator becomes smaller than machine precision even for small systems, resulting in immense numerical errors. The ε has to be strictly smaller[5] than $(1/4)^{|S|}$ [18]. According to IEEE 754.2019 standard,[6] the commonly used double machine precision is 10^{-16}, so already for 27 states the necessary ε becomes smaller than machine precision. Thus, the transformation to a stopping game is inherently impractical.

Our approach introduces additional constraints for every maximal end component (MEC) to ensure that the QP finds the correct solution.

[5] It actually has to be a lot smaller, since the *value* of every state may differ by at most that amount, but this conservative upper bound suffices to prove our point.

[6] https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html.

Algorithm 3 Algorithm to add constraints for MECs containing states of both players.

```
procedure ADD_MEC_CONSTRAINTS(MEC T ⊆ S)
    for all strategies σ on T_□ do
        for all strategies τ on T_○ do
            for every (s, a) ∈ E^T do
                Compute p^{σ,τ}_{e(s,a)}(t) for all t ∈ T by solving the modified induced MC G^{σ,τ}
    for every t ∈ T do
        Add constraint: V(t) = max_σ min_τ Σ_{(s,a)∈E^T} p^{σ,τ}_{e(s,a)}(t) · V(s, a)
```

For MECs where all states belong to the same player, the solution is straightforward. All states in MECs with only Minimizer states have a value of 0, as they can choose to remain forever in the EC and not reach the target; they can be identified a priori and are part of the set of sinks Z. All states in MECs with only Maximizer states have the value of the best exit from that MEC [32]. Thus, for all $T \in \mathsf{MEC}(\mathsf{G})$ with $T \cap S_{\bigcirc} = \emptyset$ we can introduce an additional constraint: $\forall \mathsf{s} \in T : \mathsf{V}(\mathsf{s}) = \mathsf{best\_exit}_{\mathsf{V}}(T)$, where best_exit is defined as for BVI (see Section 3.1). Note that max-constraints are expressed through continuous and boolean variables and therefore, the resulting quadratic program is a mixed-integer program.

For MECs containing states of both players, the values of the states depend on the best exit that Maximizer can ensure reaching against the optimal strategy of Minimizer. We cannot just set the value to the best exit of the whole MEC, as Minimizer might prevent some states in the MEC from reaching that best exit. The solution of [32] to analyse the graph and figure out Minimizer's decisions on the fly is not possible, because we have to give the constraints a priori.

We solve the problem as follows: iterate over all strategy-pairs (σ, τ) in the MEC and for each pair describe the corresponding value of every state $\mathsf{V}_{\sigma,\tau}(\mathsf{s})$ depending on the values of the exiting actions $E^T = \{(\mathsf{s}, \mathsf{a}) \mid \mathsf{s} \in T \wedge \mathsf{Post}(\mathsf{s}, \mathsf{a}) \not\subseteq T\}$. Then constrain the value for all states in the MEC according to the optimal strategies, i.e. $\mathsf{V}(\mathsf{s}) = \max_{\sigma} \min_{\tau} \mathsf{V}_{\sigma,\tau}(\mathsf{s})$. This ensures that the value of every state is set to the best exit it can reach, because the optimal strategies are chosen.

It remains to define how to describe $\mathsf{V}_{\sigma,\tau}$ depending on the exits E^T. For a pair of strategies (σ, τ) in the MEC, we consider the induced Markov chain $\mathsf{G}^{\sigma,\tau}$. We modify the MC and let every state-action pair $(\mathsf{s}, \mathsf{a}) \in E^T$ lead to a sink state $e_{(\mathsf{s},\mathsf{a})}$. Then, for every such sink state, we compute the probability $p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t)$ to reach it from every state $t \in T$ by solving the MC. Then we set $\mathsf{V}_{\sigma,\tau}(t) = \sum_{(\mathsf{s},\mathsf{a}) \in E^T} p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t) \cdot \mathsf{V}(\mathsf{s}, \mathsf{a})$. We summarize the procedure we just described in Algorithm 3. We use *all strategies on* T to denote every possible mapping that maps every state $t \in T$ to some available action $a \in \mathsf{Av}(t)$.

Note that $\mathsf{V}(\mathsf{s}, \mathsf{a})$ is defined as usual (see Equation (2)). As (s, a) is an exit from T, it depends on some successor $\mathsf{s}' \notin T$. Thus the states in the MEC do not depend only on each other any more, but they depend on exiting actions. Intuitively, this ensures there is a unique solution.

Lemma 4. *Let* G *be a non-stopping SG. The QP from Section 3.3.1 still is correct, i.e. returns the value vector of* G*, if we modify it as follows: For every MEC* T*, we add the constraint* $\mathsf{V}(\mathsf{s}) = \max_{\sigma} \min_{\tau} \mathsf{V}_{\sigma,\tau}(\mathsf{s})$ *for all strategies* (σ, τ) *in* T *and all states* $\mathsf{s} \in T$*.*

Proof. We proceed in the following steps:

- Summarize the relevant definitions.
- Prove that $\mathsf{V}_{\sigma,\tau}(t) = \mathbb{P}_t^{\sigma,\tau}(\Diamond F)$ for all $t \in T$.
- Prove that $\mathsf{V}(\mathsf{s}) = \max_{\sigma} \min_{\tau} \mathsf{V}_{\sigma,\tau}(\mathsf{s})$ for all strategies (σ, τ).
- Prove that adding this constraint ensures the convergence of the QP to a unique correct solution.
- Show why we can restrict to consider only strategies in the MEC T.

1. As before, we consider an SG G. We use T to denote an arbitrary MEC of G, respectively. For strategies, we write σ for Maximizer and τ for Minimizer strategies.
   The proof relies on the following definitions:
   - $E^T = \{(\mathsf{s}, \mathsf{a}) \mid \mathsf{s} \in T \wedge \mathsf{Post}(\mathsf{s}, \mathsf{a}) \not\subseteq T\}$ is the set of exiting state-action pairs.
   - For every $(\mathsf{s}, \mathsf{a}) \in E^T$, we introduce a new sink state $e_{(\mathsf{s},\mathsf{a})}$.
   - $p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t)$ is the probability to reach such a sink state $e_{(\mathsf{s},\mathsf{a})}$ from state t in the induced Markov chain $\mathsf{G}^{\sigma,\tau}$ modified so that the exiting actions lead to the newly introduced sinks.
   - Note that the symbol V is overloaded in the context of the QP: it can refer both to the value of the SG as well as to the variable of the QP that eventually converges to the value, but that can have other valuations during the computation.
     To be precise, in this proof we distinguish the actual value of the SG – V – and the value of the SG assuming we play (σ, τ) in the MEC – V' – which is what the variables of the QP are set to. However, when adding the constraint to the optimization problem, the distinction between V and V' is not necessary, since the valuation of the variable $\mathsf{V}(\mathsf{s}, \mathsf{a})$ of the QP always depends on the current strategies.

- $\mathsf{V}_{\sigma,\tau}(t) = \sum_{(\mathsf{s},\mathsf{a})\in E^T} p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t)\cdot \mathsf{V}'(\mathsf{s},\mathsf{a})$ is the value of a state $t \in T$, assuming strategies σ and τ are played.

2. We now prove that $\mathsf{V}_{\sigma,\tau}(t) = \mathbb{P}^{\sigma,\tau}_t(\Diamond F)$ for all $t \in T$. In other words, we prove that the computation we use for $\mathsf{V}_{\sigma,\tau}$ actually captures the concept of probability to reach the target under the strategies (σ,τ). For this, we use the following chain of equations:

$$\begin{aligned}
\mathsf{V}_{\sigma,\tau}(t) &:= \sum_{(\mathsf{s},\mathsf{a})\in E^T} p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t)\cdot \mathsf{V}'(\mathsf{s},\mathsf{a}) && \text{(By definition of } \mathsf{V}_{\sigma,\tau}(t)\text{)}\\
&= \sum_{(\mathsf{s},\mathsf{a})\in E^T} p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t)\cdot \left(\sum_{\mathsf{s}'\in \mathsf{Post}(\mathsf{s},\mathsf{a})} \delta(\mathsf{s},\mathsf{a},\mathsf{s}')\cdot \mathsf{V}'(\mathsf{s}')\right) && \text{(Unfolding the Bellman equation)}\\
&= \sum_{(\mathsf{s},\mathsf{a})\in E^T} p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t)\cdot \left(\sum_{\mathsf{s}'\in \mathsf{Post}(\mathsf{s},\mathsf{a})} \delta(\mathsf{s},\mathsf{a},\mathsf{s}')\cdot \mathbb{P}^{\sigma,\tau}_{\mathsf{s}'}(\Diamond F)\right) && \text{(By definition of } \mathsf{V}'\text{)}\\
&= \mathbb{P}^{\sigma,\tau}_t(\Diamond F)
\end{aligned}$$

In the last step, we pull together the unfolded probability of the path (going to some exit, taking an exiting action and then continuing from the successor); and we use the fact that all paths reaching the target have to pass through some exiting state-action pair, as otherwise they are stuck in the EC forever.
Note that this relies on the assumption that there is no target state in the MEC; this assumption is justified, since we argued in the preliminaries that every target state has only one action which is a self loop, and we exclude these trivial MECs from consideration, because their value is immediately set correctly to 1.

3. It follows from the previous step that for all $t \in T$ we have

$$\mathsf{V}(t) := \sup_\sigma \inf_\tau \mathbb{P}^{\sigma,\tau}_t(\Diamond F) = \sup_\sigma \inf_\tau \mathsf{V}_{\sigma,\tau}(t)$$

We overload the symbol $\mathsf{V}_{\sigma,\tau}$ to also denote the probability for states outside the MEC T to reach the targets under strategies σ and τ, so formally: $\mathsf{V}_{\sigma,\tau}(t) := \begin{cases} \sum_{(\mathsf{s},\mathsf{a})\in E^T} p^{\sigma,\tau}_{e_{(\mathsf{s},\mathsf{a})}}(t)\cdot \mathsf{V}'(\mathsf{s},\mathsf{a}) & \text{if } t \in T \\ \mathbb{P}^{\sigma,\tau}_t(\Diamond F) & \text{otherwise} \end{cases}$.
Then, trivially, we also have that for all state $\mathsf{s} \in S$

$$\mathsf{V}(\mathsf{s}) = \sup_\sigma \inf_\tau \mathsf{V}_{\sigma,\tau}(\mathsf{s})$$

As there are only finitely many memoryless deterministic strategies, and those strategies suffice to attain the optimal value in simple stochastic games, we also have for all state $\mathsf{s} \in S$

$$\mathsf{V}(\mathsf{s}) = \max_\sigma \min_\tau \mathsf{V}_{\sigma,\tau}(\mathsf{s})$$

4. The problem of MECs is that in these state sets there are multiple solutions to the Bellman equations (cf. Example 2), and thus multiple solutions to the quadratic program. By constraining all states in the MECs to $\max_\sigma \min_\tau \mathsf{V}_{\sigma,\tau}(\mathsf{s})$, we constrain them to exactly their value (by the previous step). Thus, we solve the problem, as now the additional solutions are excluded.
5. Note that so far, this proof considered strategies on the whole state space. However, our algorithm only iterates over all strategies *in the MEC*.
This is sufficient, because the states outside the MEC are solved by the rest of the QP, and the newly added constraints depend on those solutions, as they depend on the valuation of $\mathsf{V}(\mathsf{s},\mathsf{a})$ of exiting state-action pairs. □

For a MEC of size n we have to examine at most $(\max_{\mathsf{s}\in T}|\mathsf{Av}(\mathsf{s})|)^n$ pairs of strategies, because it suffices to consider memoryless deterministic strategies. Since the min and max constraints have to be explicitly encoded, we have to add a number of constraints that is exponential in the size of the MEC. Thus, in the worst case, this reduction creates a QP of size exponential in the size of SG, which is worse than the original reduction using the normal form. However, as MECs are typically small, the approach still is more practical.

4.1.5. Summary

In summary, we have shown how to replace every condition of the normal form by modifying the constraints and objective function of the program. The modifications always ensure that the program still computes the correct value, because it still only has a single global optimum in the constrained region, namely if every state variable is set to its value. Thus, we can provide a higher-order program to solve arbitrary SGs, and a quadratic program to solve SGs that only have to satisfy the **2Act** condition.

4.2. Opponent strategy for strategy iteration

We can tune the MDP solution method that is used to compute the opponent strategy in Line 6 of Algorithm 2. We need to ensure that we fix the new choices of Maximizer correctly. For this, we can use the precise MDP solution methods strategy iteration or linear programming (LP, a QP with an objective function of order 1). However, as [34] pointed out for the mean payoff objective, we do not need the precise solution of the induced MDP, but it suffices to know that an action is better than all others. So we can also use bounded value iteration and check that the lower bound of one action is larger than the upper bound of all other actions, and thus we can stop the algorithm earlier. This approximation and the fact that VI tends to be the fastest methods in MDPs can speed up the solving. Using unguaranteed VI is dangerous, as it might stop too early and return a wrong strategy.

4.3. Warm start

All three solution methods can benefit from prior knowledge. VI and quadratic/higher-order programs (QP/HOP) can immediately use initial solution vectors obtained by domain knowledge or any precomputation. For VI, it is necessary to know whether it is an upper or lower estimate to use the prior knowledge correctly. The QP/HOP optimization process can start at the given initial vector.

SI can use the information of an initial estimate to infer a good initial strategy, as already suggested in [34]. This reduces the number of iterations of the main loop and thus the runtime. However, we have to ensure that the resulting strategy is proper. For example, we can check whether the target and sink states are reached almost surely from every state, and if not, we change the strategy to an attractor strategy where necessary, preferring those allowed actions that have a higher value.

We can also improve the MDP-solving for SI (Line 6 of Algorithm 2) by giving it the knowledge we currently have. We anyway save the estimate L of the previous iteration and, since the strategies of Maximizer get monotonically better, L certainly is a lower bound for the values in the MDP.

Even in the absence of domain knowledge or sophisticated precomputations, we can run unguaranteed VI first in order to get some estimates of the values. Then we can use those estimates for SI and QP/HOP. Note that this is similar in spirit to the idea of optimistic value iteration [28]: utilize VI's ability to usually deliver tight lower bounds and then verify them.

4.4. Topological improvement

For MDPs, topological improvements have been proposed for VI [22] and for SI [34, Algorithm 3]. These utilize the fact that the underlying graph of the MDP can be decomposed into a directed acyclic graph of strongly connected components (SCC). Intuitively speaking, there are parts of the graph that, after leaving them, can never be reached again. Their value depends solely on the parts of the state space that come after them. So instead of computing the values on the whole game at once, one can iterate over the SCC in a backwards fashion, starting with the target and sink states and then propagating the information and solving the SCCs one by one according to their topological ordering.

This idea was extended to BVI for MDPs in [4]. The proof generalizes to SGs, as the basic argument of the topological ordering of SCCs is independent from introducing a second player. As the proof relies on the solutions for the later components being ε-precise, solving those components with the precise methods SI or QP is of course also possible.

5. Experimental comparison

First, we give all the meta information about the experimental setup and the way we describe the results in Section 5.1. Then, in Section 5.2 to 5.4 we compare different variants of the same algorithm, including a detailed look at the behaviour of QP on models with many actions. Finally in Section 5.5 we compare the between the algorithm classes, considering runtime, memory and the bigger theoretical context.

5.1. Experimental setup

5.1.1. Implementation

We implemented all our algorithms as an extension of PRISM-games 3.0 [36]. The implementation is available via github.[7] For BVI, we reimplemented the algorithm as described in [32]; for SI and QP, this is the first implementation in PRISM-games. To solve the quadratic program, we used Gurobi[8] or CPLEX.[9] For the higher-order programs, we constructed them with AMPL and solved them with MINOS.[10]

[7] https://github.com/ga67vib/Algorithms-For-Stochastic-Games.
[8] https://www.gurobi.com/.
[9] https://www.ibm.com/analytics/cplex-optimizer.
[10] https://ampl.com/products/solvers/solvers-we-sell/minos/.

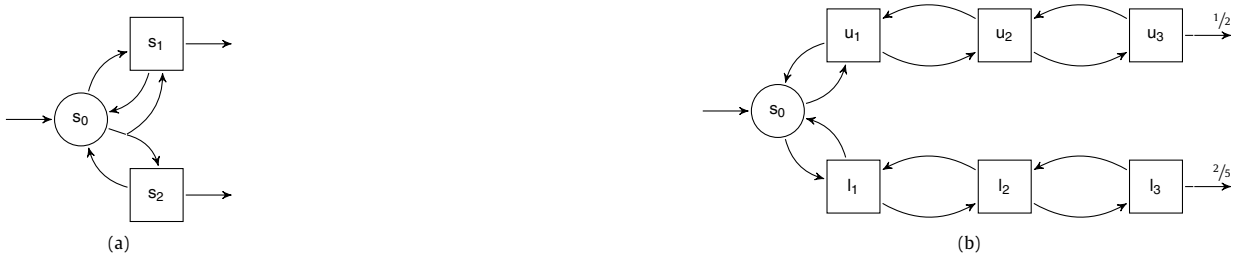

Fig. 5. Fig. 5a depicts the MEC that is used in the handcrafted MulMec model, as well as in cdmsnMEC and dice50MEC. Fig. 5b shows the handcrafted scalable model "BigMec" with $N = 3$. In this model, there are $2 \cdot N + 1$ states in the MEC and a dedicated target and sink. The initial state is a Minimizer state, leading to two chains of length N of Maximizer states. The fractions on the leaving actions denote the values; they are obtained by a transition to the target state with the given probability and to the sink with the remaining probability. The value of every state in the upper chain is 0.5; the initial state and the lower chain have value 0.4.

5.1.2. Setup

All experiments were conducted on a Linux Manjaro server with a 3.60 GHz Intel(R) Xeon(R) W-2123 CPU and 64 GB of RAM. We used a timeout of 15 minutes and set the java heap size for PRISM-games to 32 GB and the stack size to 16 GB.[11] The precision for BVI was set to 10^{-6}. In theory, the other algorithms are precise. In practice, QP and higher-order programming can have numerical problems; we mark these cases with red background colour in the table (this is only visible in the web version of the article).

For SI, even when using strategy iteration to solve the MDP of the opponent, PRISM still solves the resulting Markov chain by value iteration. For all of the models except hm, our solution is still precise, because we detect and fix probabilistic cycles of size 1. The model hm is the only one in our benchmark set that has larger probabilistic cycles that cause issues with convergence. However, this implementation detail does *not* imply that SI is in general not precise. For example, solving the Markov chain with linear programming would result in precise solutions.

5.1.3. Case studies

On the one hand, we used all case studies we are aware of that model a realistic scenario as a simple stochastic game. On the other hand, we generated some examples of interesting theoretical extreme cases ourselves, to see the behaviour the algorithms exhibit and estimate the impact of certain subcomponents.

As realistic case studies, we use the ones that are distributed with PRISM-games 3.0 and available on their case-study-website.[12] These are coins, prison_dil, adt, charlton, cdmsn, cloud, mdsm, dice and two_investors. Moreover, we use the HW and AV models which were introduced in [13]. The first index indicates the size of the square grid a robot moves on, the second index denotes the property that is checked.

As handcrafted models, we prepended the MEC-free models dice and cdmsn with a single small MEC (depicted in Fig. 5a) in order to judge the impact of such a single MEC. The exits lead to the initial state of the original model with some probability, and the remaining probability leads to a sink. Further, we used the adversarial model for value iteration from [25] (called hm[13]) as well as two newly handcrafted models with either one large MEC (BigMec) or many 3 state MECs (MulMec). In BigMec, there is a MEC with two chains of N Maximizer states, see Fig. 5b. In MulMec, a single MEC is repeated N times. For the first N-1 repetitions, both exits lead to s_0 of the next MEC with some probability and to s_0 of the current MEC with the rest. For the last repetition, both exits lead to some probabilistic combination of target and sink.

Note that the largest models we use in the paper have less than 500,000 states, so we can only make limited statements about how the algorithms scale when applied to extremely large, realistic SGs. A reason for few benchmarks being available probably is that so far there were no guaranteed solvers for simple stochastic games. We hope that our implementation will motivate others to model their problems as SGs and thus enable further analysis on realistic case studies. Moreover, we discuss at the end of Section 5.5 that structural properties of a model are more important than its size. This indicates that to make claims about scalability to extremely large models, we also need to take into account their structural properties.

5.1.4. Table layout and considered algorithms

We first analyse the impact of our optimizations by comparing different variants of the same algorithm: value iteration in Section 5.2 and Table 1; strategy iteration in Section 5.3 and Table 2; and quadratic respectively higher-order programming in Section 5.4 and Table 3. Based on this, we select the same variants of each algorithm and compare between the algorithms in Section 5.5; here we consider both runtime in Table 4 and memory in Table 5 and comment on the context of the theoretical properties of the algorithms.

[11] -javamaxmem 32g -javastack 16g.

[12] http://www.prismmodelchecker.org/games/casestudies.php.

[13] hm is short for haddad-monmege, the names of the authors of [25].

Every runtime table includes the verification times (in seconds) of several variations of the algorithms, i.e. different optimizations enabled or disabled. An X in the table denotes that the computation did not finish within 15 minutes. A red background colour indicates that the returned result was wrong, i.e. off by more than the allowed precision (the red background colour is only visible in the web version of this article). All results that are wrong are off significantly, i.e. by more than 0.1. The four left-most columns are shared by all tables and give relevant properties that are interesting when analysing the performance. They show the name of the considered case study, its size, the maximum/average number of actions and the number of MECs; for the latter, note that this number excludes trivial MECs, i.e. sink or target states and MECs in regions of the graph that are either not reachable or identified as trivially having value 0/1 by the precomputation. The case studies are roughly sorted by increasing size/difficulty, with scaled versions of the same model grouped together. For the comparison within the algorithms, we report all case studies we used; for the comparison between the algorithms, we restricted to the most interesting ones.

We consider the following algorithms:

VI denotes unguaranteed value iteration.
BVI denotes bounded value iteration.
Sim denotes the simulation-based variant of value iteration [32]. As this is a randomized algorithm, we report both the maximum and median runtime of three tries.
SI denotes strategy iteration.
[19] is the original QP algorithm that transforms the SG into normal form.
[19]$_\varepsilon$ is a small modification of the original algorithm that only introduces the ε transition to make it stopping if necessary.
QP denotes quadratic programming for arbitrary SGs, only transforming the SG to satisfy the **2Act**-constraint.
HOP is higher-order programming for arbitrary SGs.

For each algorithm, optimizations are indicated as follows:

T as a prefix denotes the topological optimization.
W as a superscript denotes the warm start optimization, i.e. unguaranteed value iteration to guess a good initial strategy.
D in the subscript of BVI denotes an optimization suggested in [32]: instead of "deflating" (finding SEC candidates and decreasing their over-approximation) in every step, this costly operation is only carried out every 100 steps.
SI/BV in the subscript of SI indicates the method used to solve the MDP, either SI or BVI.
C/G in the subscript of QP indicates the used solver, C for CPLEX and G for Gurobi.

5.2. Value iteration (Table 1)

We now analyse the results of the different variants of value iteration. We could reproduce most of the findings of [32]: the overhead of BVI compared to unguaranteed VI is usually negligible and not performing the expensive deflate operation in every step speeds up the computation. Unguaranteed VI fails on three of the models. In contrast to [32], we found no model where the simulation based asynchronous version BRTDP was significantly faster than BVI. In fact, BRTDP is only faster on a single model (cloud6, which is not too significant as BVI also takes only 2 seconds), but significantly slower on many others, often even failing to produce results in time. Note that the implementation of BRTDP was in PRISM-games 2, and thus the disadvantage may also have technical reasons; improvements in the simulation engine or data structures might lead to speed-ups that make BRTDP competitive again.

The new topological variant of BVI (called TBVI) is usually in the same order of magnitude as the default approach. The exception to this are the models AV15_15 and especially MulMec, where TBVI fails. This is because TBVI solves every SCC of the model with a precision of ε. When one SCC is solved and has a difference of almost exactly ε between upper and lower bound, SCCs depending on it have suboptimal information about their exits. This not only slows down convergence, but can even aggregate and lead to precision problems when there is a chain of many SCCs. Indeed, for MulMec with N$>$15, the progress that TBVI makes in the 15th SCC it considers is smaller than machine precision, and thus the computation is stuck. This problem is not specific to topological VI for SGs, but can also occur for MDPs.

5.3. Strategy iteration (Table 2)

We now analyse the results of the different variants of strategy iteration. Using BVI for the opponent's MDP and the warm start usually lead to small speed-ups. We did not consider using linear programming for the opponent's MDP, as it is not supported by PRISM. The topological variant is significantly faster in two_inv and MulMec_e3, but on the rest of the models performs very similar to the non-topological version. Combining topological SI and BVI for the opponent's MDP leads to the same problems with MulMec as when using topological BVI. In contrast, topological SI with SI for the opponent's

Table 1
Table with all experimental results on variations of value iteration. See Section 5.1.4 for a description of table layout and considered algorithms, and Section 5.2 for the analysis.

Case Study	States	Acts	MECs	VI	Sim	BVI	BVI_D	TBVI	$TBVI_D$
coins	19	2 | 1.1	0	<1	<1	<1	<1	<1	<1
prison_dil	102	3 | 1.3	0	<1	<1	<1	<1	<1	<1
adt	305	4 | 1.2	0	<1	X	<1	<1	<1	<1
charlton1	502	3 | 1.5	0	<1	<1	<1	<1	<1	<1
charlton2	502	3 | 1.5	0	<1	<1	<1	<1	<1	<1
cdmsn	1,240	2 | 1.6	0	<1	<1	<1	<1	<1	<1
cdmsnMec	1,244	2 | 1.6	1	<1	<1	<1	<1	<1	<1
cloud5	8,842	11 | 3.9	520	<1	9 | 6	<1	<1	<1	<1
cloud6	34,954	13 | 4.4	2176	<1	<1	2	2	8	3
mdsm1	62,245	2 | 1.3	0	4	7 | 7	5	5	3	3
mdsm2	62,245	2 | 1.3	0	<1	15 | 14	<1	<1	<1	<1
dice20	16,915	2 | 1.4	0	<1	262 | 222	<1	<1	<1	<1
dice50	96,295	2 | 1.4	0	6	X	6	6	6	6
dice50Mec	96,299	2 | 1.4	1	6	X	6	6	6	6
two_inv	172,240	3 | 1.3	0	13	X	14	13	13	13
hw5_1	25,000	5 | 2.4	0	<1	87 | 18	<1	<1	<1	<1
hw5_2	25,000	5 | 2.4	0	<1	<1	<1	<1	<1	<1
hw8_1	163,840	5 | 2.5	0	3	X	3	3	3	3
hw8_2	163,840	5 | 2.5	0	<1	<1	<1	<1	<1	<1
hw10_1	400,000	5 | 2.5	0	10	X	10	10	10	11
hw10_2	400,000	5 | 2.5	0	<1	<1	<1	<1	1	1
AV10_1	106,524	6 | 2.1	0	<1	X	<1	<1	<1	<1
AV10_2	106,524	6 | 2.1	6	70	X	81	72	82	70
AV10_3	106,524	6 | 2.1	1	47	X	46	45	52	55
AV15_1	480,464	6 | 2.1	0	1	X	1	1	1	1
AV15_2	480,464	6 | 2.1	6	729	X	X	X	X	X
AV15_3	480,464	6 | 2.1	1	492	X	X	X	X	X
hm_30	61	1 | 1.0	0	<1	X	X	X	X	X
MulMec_e2	302	2 | 1.9	100	<1	X	<1	2	X	X
MulMec_e3	3,002	2 | 2.0	1000	4	X	36	151	X	X
MulMec_e4	30,002	2 | 2.0	10000	591	X	X	X	X	X
BigMec_e2	203	2 | 1.9	1	<1	X	<1	<1	<1	<1
BigMec_e3	2,003	2 | 2.0	1	<1	X	6	1	6	1
BigMec_e4	20,003	2 | 2.0	1	161	X	856	226	871	230

MDP works, because SI solves the SCCs precisely, allowing the SCCs depending on the previous solutions to converge as well.

5.4. Quadratic programming

5.4.1. Runtime comparison (Table 3)

We now analyse the results of the different variants of quadratic or higher-order programming. The original version of quadratic programming [19], that requires to transform the SG into normal form is impractical, producing a result within 15 minutes for only 3 of 34 case studies, and even there taking a lot more time than the improved version. After dropping the constraints **No1Act** and **½Probs**, it can correctly solve 14 of the case studies in time. Both of these variants are exhibit the numerical errors described in Section 4.1.4 and produce incorrect results on some models. The quadratic program obtained after dropping all but the **2Act** constraint is solved successfully by Gurobi in 19 instances; CPLEX on the other hand only solves 7 instances correctly, one time even reporting an incorrect result. The warm start helps Gurobi on the model charlton1. Several other times, Gurobi's internal heuristics are deemed more promising by the program and it discards the given initial suggestion; thus, the warm start sometimes incurs a slight overhead.

Dropping all constraints, the higher-order program (HOP) with the solver Minos gets the correct result on 26 instances. The HOP approaches are typically faster than the QP ones, except on the models HW and AV. The topological variant of both the quadratic programs as well as the higher-order program can lead to significant speed-ups, for example on charlton1, cloud5 and cloud6, mdsm1, two_inv or HW10_10_2. On all models but cloud, the topological HOP is strictly faster than all other algorithms in this subsection.

To estimate the impact of the EC solution method, we used several handcrafted or modified models: A single large MEC (BigMec_e2, with a MEC of size 201) cannot be solved with our approach, as there are too many choices; possibly, some heuristic could help identify reasonable strategies. In contrast, small MECs do not affect runtime a lot. The models cdmsn

Table 2
Table with all experimental results on variations of strategy iteration. See Section 5.1.4 for a description of table layout and considered algorithms, and Section 5.3 for the analysis.

Case Study	States	Acts	MECs	SI_{SI}	SI_{BV}	SI^{W}_{SI}	SI^{W}_{BV}	TSI_{SI}	TSI_{BV}	TSI^{W}_{SI}	TSI^{W}_{BV}
coins	19	2 | 1.1	0	<1	<1	<1	<1	<1	<1	<1	<1
prison_dil	102	3 | 1.3	0	<1	<1	<1	<1	<1	<1	<1	<1
adt	305	4 | 1.2	0	<1	<1	<1	<1	<1	<1	<1	<1
charlton1	502	3 | 1.5	0	<1	<1	<1	<1	<1	<1	<1	<1
charlton2	502	3 | 1.5	0	<1	<1	<1	<1	<1	<1	<1	<1
cdmsn	1,240	2 | 1.6	0	<1	<1	<1	<1	<1	<1	<1	<1
cdmsnMec	1,244	2 | 1.6	1	<1	<1	<1	<1	<1	<1	<1	<1
cloud5	8,842	11 | 3.9	520	<1	<1	<1	<1	<1	<1	<1	<1
cloud6	34,954	13 | 4.4	2176	<1	<1	<1	<1	<1	<1	<1	<1
mdsm1	62,245	2 | 1.3	0	5	5	6	6	3	3	3	3
mdsm2	62,245	2 | 1.3	0	<1	<1	<1	<1	<1	<1	<1	<1
dice20	16,915	2 | 1.4	0	<1	<1	<1	<1	<1	<1	<1	<1
dice50	96,295	2 | 1.4	0	7	6	7	7	6	6	6	6
dice50Mec	96,299	2 | 1.4	1	6	7	7	7	6	6	6	6
two_inv	172,240	3 | 1.3	0	38	19	37	22	11	11	13	12
HW5_1	25,000	5 | 2.4	0	<1	<1	<1	<1	<1	<1	<1	<1
HW5_2	25,000	5 | 2.4	0	<1	<1	<1	<1	<1	<1	<1	<1
HW8_1	163,840	5 | 2.5	0	4	4	4	4	4	4	4	4
HW8_2	163,840	5 | 2.5	0	<1	<1	<1	<1	<1	<1	<1	<1
HW10_1	400,000	5 | 2.5	0	11	11	11	11	10	11	11	11
HW10_2	400,000	5 | 2.5	0	2	2	2	2	1	1	2	2
AV10_1	106,524	6 | 2.1	0	<1	<1	<1	<1	<1	<1	<1	<1
AV10_2	106,524	6 | 2.1	6	83	79	79	80	79	76	77	79
AV10_3	106,524	6 | 2.1	1	55	50	58	52	60	X	60	X
AV15_1	480,464	6 | 2.1	0	2	2	2	2	1	2	2	2
AV15_2	480,464	6 | 2.1	6	797	825	843	791	X	X	X	X
AV15_3	480,464	6 | 2.1	1	529	500	512	497	X	X	X	X
hm_30	61	1 | 1.0	0	X	X	X	X	X	X	X	X
MulMec_e2	302	2 | 1.9	100	<1	X	<1	X	<1	X	<1	X
MulMec_e3	3,002	2 | 2.0	1000	56	X	56	X	3	X	3	X
MulMec_e4	30,002	2 | 2.0	10000	X	X	X	X	X	X	X	X
BigMec_e2	203	2 | 1.9	1	<1	<1	<1	<1	<1	<1	<1	<1
BigMec_e3	2,003	2 | 2.0	1	1	1	1	1	1	1	1	1
BigMec_e4	20,003	2 | 2.0	1	X	X	X	X	X	X	X	X

and dice50 prepended with a single three-state MEC are solved in the same time as the original models. Even a chain of 1000 three-state MECs can be solved quickly (MulMec_e3).

5.4.2. Additional analysis: impact of the number of actions (Fig. 6)

It seems plausible that a high number of actions is problematic for the QP/HOP-algorithms, since for QP the SG has to be blown up and for HOP the degree of the objective function increases. Thus, independent of the runtime comparison, in this subsection we investigate how the number of actions per state affects the performance of the algorithms.

The highest number of actions per state in the real-word case studies is 13 and the highest average is 4.4. We want to analyse the performance with even higher number of actions and on more models. However, the PRISM language, which is used for encoding the models, does not have the feature of scalable number of actions. Hence, we used random generation to obtain models with a large number of actions per state.

Random generation of models: We have the following requirements for the randomly generated models: firstly they should not be trivial, so we filter out models that are solved by standard precomputations or have value 1. Secondly, all models should be of the same size, to ensure that effects we see are not due to an increase in size, but rather due to increasing the number of actions. Similarly, the models should not contain non-trivial MECs, as we have already seen that these strongly influence on the runtime of the algorithms. Their presence could obfuscate the impact of the number of actions.

We use two different approaches to generate the random models, where in both, every state has equal chance of being a Maximizer or Minimizer state; also, both are parametrized by the designated number of actions per state m.

- Random tree: The main underlying topology of this model is a tree. That means that every inner node has m actions that have a non-zero probability of leading to states of the next level. Additionally, there may be transitions to random other states of the tree, where the smallest transition probability is 0.1. The generated tree can consist of a single SCC – if all states have some path back to the root – or of as many SCCs as there are states in the tree – if no state has a transition to an ancestor. Every state has a non-zero probability of reaching a leaf state, which are either targets or sinks. Thus, there can be no non-trivial MECs. We report our results for tree size 1000, because QP could still solve several models of this size. For larger trees, it regularly timed out, i.e. needs more than 15 minutes.
  For every $m \in \{2, 5, 10, 30, 50, 70, 90, 100\}$, we generated 20 trees, resulting in a total of 160 models.

Table 3
Table with all experimental results on variations of quadratic/higher order programming. See Section 5.1.4 for a description of table layout and considered algorithms, and Section 5.4 for the analysis.

Case Study	States	Acts	MECs	[19]	$[19]_\varepsilon$	QP_C	QP_G	QP_G^W	HOP	TQP_G	TQP_G^W	THOP
coins	19	2 | 1.1	0	X	X	<1	<1	<1	<1	<1	<1	<1
prison_dil	102	3 | 1.3	0	X	8	11	9	8	<1	<1	<1	<1
adt	305	4 | 1.2	0	X	<1	<1	<1	<1	<1	<1	<1	<1
charlton1	502	3 | 1.5	0	X	179	X	180	144	<1	<1	<1	<1
charlton2	502	3 | 1.5	0	X	X	X	X	X	<1	X	X	<1
cdmsn	1,240	2 | 1.6	0	17	<1	<1	<1	<1	<1	<1	<1	<1
cdmsnMec	1,244	2 | 1.6	1	X	<1	<1	<1	<1	<1	<1	<1	<1
cloud5	8,842	11 | 3.9	520	X	X	X	X	X	1	3	3	4
cloud6	34,954	13 | 4.4	2176	X	X	X	X	X	8	14	15	21
mdsm1	62,245	2 | 1.3	0	X	X	X	X	X	107	6	7	3
mdsm2	62,245	2 | 1.3	0	X	<1	1	<1	<1	5	<1	<1	<1
dice20	16,915	2 | 1.4	0	X	3	X	3	3	2	<1	<1	<1
dice50	96,295	2 | 1.4	0	X	X	X	X	X	15	6	6	6
dice50Mec	96,299	2 | 1.4	1	X	X	X	X	X	12	6	7	6
two_inv	172,240	3 | 1.3	0	X	X	253	X	X	432	X	X	20
HW5_5_1	25,000	5 | 2.4	0	13	1	X	1	1	3	<1	<1	<1
HW5_5_2	25,000	5 | 2.4	0	X	1	X	1	1	3	<1	<1	<1
HW8_8_1	163,840	5 | 2.5	0	96	22	X	20	20	17	4	4	4
HW8_8_2	163,840	5 | 2.5	0	X	11	X	11	11	47	2	2	<1
HW10_10_1	400,000	5 | 2.5	0	X	98	X	98	98	44	12	12	11
HW10_10_2	400,000	5 | 2.5	0	X	48	X	49	49	286	4	4	1
AV10_10_1	106,524	6 | 2.1	0	X	3	X	3	3	9	<1	<1	<1
AV10_10_2	106,524	6 | 2.1	6	X	X	X	X	X	X	X	X	X
AV10_10_3	106,524	6 | 2.1	1	X	X	X	X	X	X	X	X	X
AV15_15_1	480,464	6 | 2.1	0	X	15	X	10	11	41	3	3	2
AV15_15_2	480,464	6 | 2.1	6	X	X	X	X	X	X	X	X	X
AV15_15_3	480,464	6 | 2.1	1	X	X	X	X	X	X	X	X	X
hm_30	61	1 | 1.0	0	735	<1	<1	<1	<1	<1	<1	<1	<1
MulMec_e2	302	2 | 1.9	100	X	X	X	<1	<1	<1	<1	<1	<1
MulMec_e3	3,002	2 | 2.0	1000	X	X	X	4	7	3	5	8	4
MulMec_e4	30,002	2 | 2.0	10000	X	X	X	X	X	X	X	X	X
BigMec_e2	203	2 | 1.9	1	X	X	X	X	X	X	X	X	X
BigMec_e3	2,003	2 | 2.0	1	X	X	X	X	X	X	X	X	X
BigMec_e4	20,003	2 | 2.0	1	X	X	X	X	X	X	X	X	X

- Repeated tree: The underlying topology of this model is a series of 20 trees, where every tree has 400 states. This ensures the problem has a certain complexity, as there are at least 20 SCCs. Every tree is generated in similar fashion as described in the random-tree-approach. For every $m \in \{2, 5, 10, 30, 50, 70, 100, 200\}$, we generated two models, resulting in a total number of 16.

Analysis: In Fig. 6a, we see the results on the random tree models. For both QP and HOP, there is no visible correlation between number of actions per state. Contrary to the claim that more actions increase the runtime, we see that most time outs occur for models with very low m.

Fig. 6b shows the results on the repeated tree models. QP times out for all of them, while HOP is able to solve all but one. Here, it looks like there is a linear relation between the number of actions per state and the runtime. Still, even for 200 actions per state, HOP is able to solve a benchmark.

In conclusion: in real case studies the average number of actions per state is typically smaller than 10. Since HOP is able to solve models with more than 100 actions per state, this model property will likely not be prohibitive. Moreover, we saw timeouts occur even for small number of actions per state, indicating that other structural properties of the models are more important.

5.5. Comparison between algorithm classes

Comparing all classes of algorithms, we differentiate three topics: First the runtime comparison, similar to the previous subsections, then a memory comparison and finally a comparison of theoretical properties to set the practical results into context.

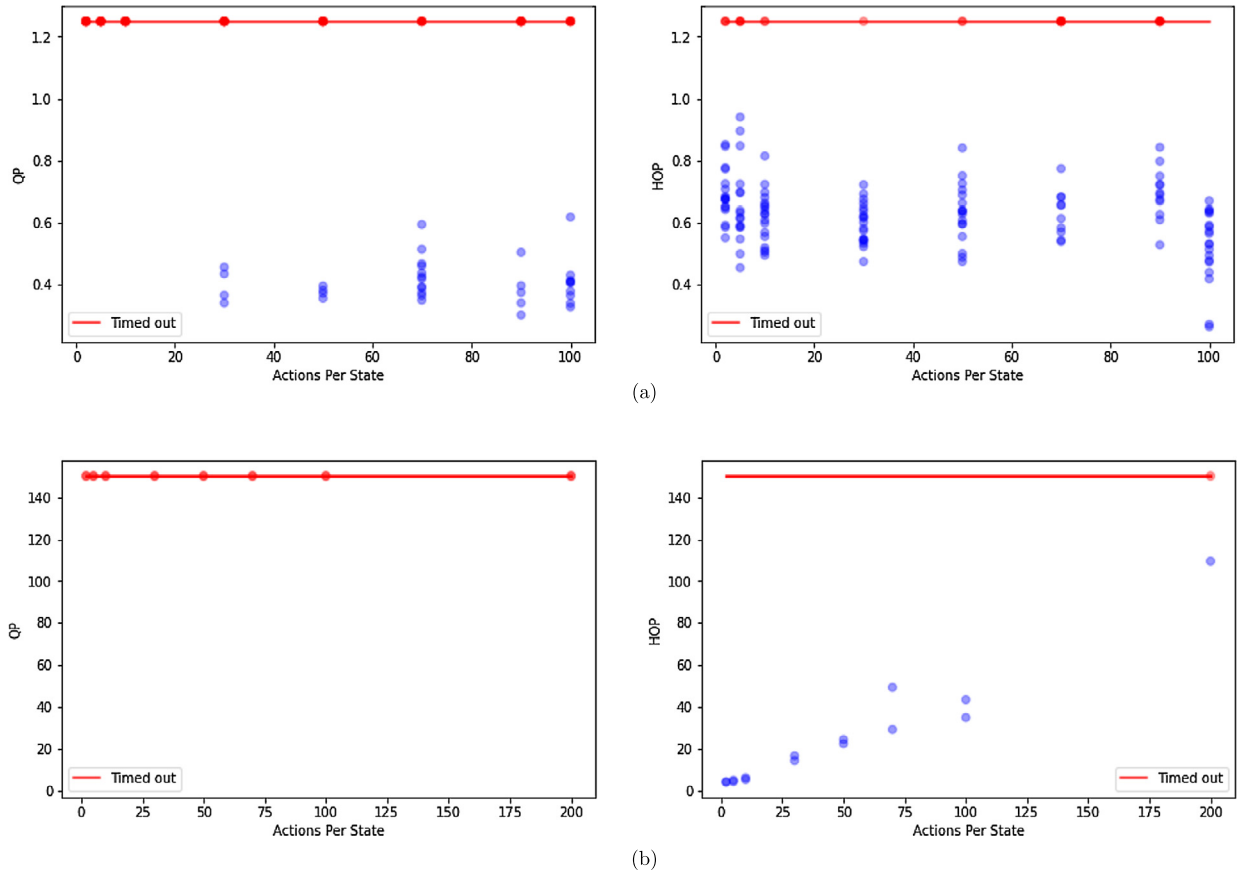

Fig. 6. Analysis of the number of actions per state on the runtime of QP and HOP. Fig. 6a and Fig. 6b show the results on random models generated by the random-tree-approach respectively the repeated-tree-approach. The y-axis of all plots denotes runtime of the algorithm in seconds. The red line denotes a time out, which we report after 15 minutes (The colour is only visible in the web version of the article).

5.5.1. Runtime comparison (Table 4 and Fig. 7)

For the runtime comparison between the different classes of algorithms, we provide the experimental data in two forms:

- In Fig. 7 we visualize the number of benchmarks solved as well as the aggregated time to solve them. We refer to the caption of Fig. 7 for a detailed description of the meaning of the lines.
- In Table 4 we present a subset of benchmarks and algorithms from the previous tables. For each algorithm we selected one variant with and one without the topological optimization, because it can have significant positive as well as negative impact. We also included one QP and one HOP, as there is a fundamental difference between the approaches. Regarding the other optimizations, we selected the ones that minimize the aggregated runtime over all benchmarks.

Fig. 7 provides an overview of the performance of all algorithms. Note that this plot immensely depends on the selection of benchmarks. By e.g. including several variants of the hm model, one could skew it and make all VI based algorithms seem worse. Still, to the best of our knowledge our set of benchmarks is somewhat balanced which allows us to extract general trends.

We see that many variants of QP and the simulation based VI variant end significantly earlier than all SI and deterministic VI algorithms. The worst algorithm is the unmodified QP variant from [19] with 3 benchmarks, while using TQP, HOP or THOP at least comes close to the SI and VI variants with 24 respectively 25 solved benchmarks. The SI and VI variants to the right of the plot all solve between 26 and 31 benchmarks. For the first 26 benchmarks they are very comparable in runtime to each other and to TQP and THOP. They only increase above 250 seconds when solving the hardest case studies.

We now turn to the more detailed analysis of Table 4. Observe that BVI and SI succeed/fail on different models. While BVI succeeds in the largest version of BigMec, SI is the only one to solve the large and complicated models AV15_15_2/3, and TSI has the best runtime in MulMec_e3. TSI benefits from a model with small subcomponents that it can quickly solve (as in MulMec), while BVI is fast in subgraphs without probabilistic cycles (as the large chains in BigMec). Depending on the model structure, both of these algorithms are a viable choice.

Table 4
Table for the runtime comparison between the classes of algorithms. See Section 5.1.4 for a description of table layout and considered algorithms, and Section 5.5.1 for the analysis.

Case Study	States	Acts	MECs	BVI_D	$TBVI_D$	SI_{BV}	TSI^W_{SI}	QP^W_G	THOP
prison_dil	102	3/1.3	0	<1	<1	<1	<1	8	<1
charlton1	502	3/1.5	0	<1	<1	<1	<1	144	<1
cdmsn	1,240	2/1.6	0	<1	<1	<1	<1	<1	<1
cdmsnMec	1,244	2/1.6	1	<1	<1	<1	<1	<1	<1
cloud6	34,954	13/4.4	2176	2	3	<1	<1	X	21
mdsm1	62,245	2/1.3	0	5	3	5	3	X	3
dice50	96,295	2/1.4	0	6	6	6	6	X	6
dice50Mec	96,299	2/1.4	1	6	6	7	6	X	6
two_inv	172,240	3/1.3	0	13	13	19	13	X	20
HW10_10_1	400,000	5/2.5	0	10	11	11	11	98	11
HW10_10_2	400,000	5/2.5	0	<1	1	2	2	49	1
AV10_10_1	106,524	6/2.1	0	<1	<1	<1	<1	3	<1
AV10_10_2	106,524	6/2.1	6	72	70	79	77	X	X
AV10_10_3	106,524	6/2.1	1	45	55	50	60	X	X
AV15_15_1	480,464	6/2.1	0	1	1	2	2	11	2
AV15_15_2	480,464	6/2.1	6	X	X	825	X	X	X
AV15_15_3	480,464	6/2.1	1	X	X	500	X	X	X
hm_30	61	1/1.0	0	X	X	X	X	<1	<1
MulMec_e2	302	2/1.9	100	2	X	X	<1	<1	<1
MulMec_e3	3,002	2/2.0	1000	151	X	X	3	7	4
MulMec_e4	30,002	2/2.0	10000	X	X	X	X	X	X
BigMec_e2	203	2/1.9	1	<1	<1	<1	<1	X	X
BigMec_e3	2,003	2/2.0	1	1	1	1	1	X	X
BigMec_e4	20,003	2/2.0	1	226	230	X	X	X	X

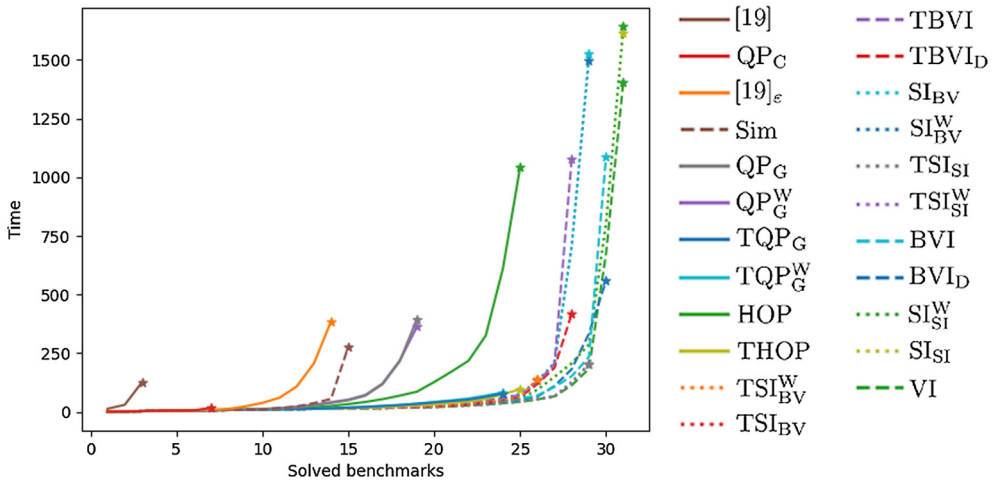

Fig. 7. An overview of the aggregated runtimes of all 23 considered algorithms. For each algorithm, we ordered the benchmarks it solved by time taken to solve them. A point (x, y) indicates that to solve the "first" x benchmarks, the algorithm had an aggregated runtime of y seconds. The star at the end of a line highlights the maximum number of benchmarks the algorithm could solve, i.e. all other case studies could not be solved within the timeout of 15 minutes. VI variants are depicted with dashed, SI with dotted and QP/HOP with solid lines. Note that the legend is ordered by performance, i.e. algorithms with less solved benchmarks (and more required time in case of ties) are listed first.

Topological higher-order programming (THOP), the improved version of QP, is comparable on many case studies, but still a lot worse on several others, e.g. cloud6 and BigMec. This volatility is even more pronounced for the QP approaches. The reason for this can be MECs which blow up the QP, but it can also happen in models with few or no MECs; in the latter case, we do not know which property of the model slows down the solving. Note that QP and HOP are able to solve models with many small MECs (MulMec_e3) quickly, while already one medium sized MEC (BigMec_e2) makes it infeasible.

The model hm_30 from [25] deserves special attention: it only is an MC, i.e. there are no nondeterministic choices. Still, as it is a handcrafted extreme case, we can observe interesting behaviour of the algorithms. Since hm is an adversarial example for VI, of course all variants of VI fail. Moreover, since the solvers for MDPs and MCs in PRISM use VI, and since SI relies on those solvers, the PRISM implementation of SI also fails on the model (see Section 5.1.2 for more details). QP shines on this model, being the only method to solve it. However, for increasing parameter N, the probabilities in hm_N become so small that they are at the border of numerical stability. If the state-chains in the model were prolonged by one

Table 5
Table for the memory comparison between the classes of algorithms. See Section 5.1.4 for a description of considered algorithms, and Section 5.5.2 for the analysis.

	BVI	TBVI	SI	[19]	$[19]_\varepsilon$	QP_C	QP_G	HOP
# min. memory used	31/34	26/34	32/34	4/34	18/34	27/34	24/34	34/34
Maximum overhead	4×	29×	2×	88×	7×	8×	3×	1×

more state (i.e. the parameter N is set to 31), QP has numerical problems and reports an incorrect result. Similarly, noting that THOP reports an incorrect result on hm_30, one can experimentally find out that THOP succeeds only for $N \leq 25$. So if the model exhibits very small probabilities, one has to consider using a solver capable of arbitrary-precision arithmetic.

Note that the size of the model is only loosely correlated with the difficulty of solving it. For example, the largest case study AV15_1 with 480,464 states can be solved in few seconds by all algorithm classes. In contrast, already the small model hm_30 with 61 states takes a long time for all VI-based algorithms and can induce precision problems for all QP-based ones. Further, the medium-sized model MulMec_e4 with 30,002 states is not solved in time by any algorithm. The reason for the difficulty of the hm model is the number of probabilistic loops as well as the small probability of the path from initial state to target. The difficulty of MulMec and BigMec models stems from the number and size of MECs. We see that these structural properties of the underlying graph have a huge effect on the performance of the algorithms.

5.5.2. Memory comparison (Table 5)

For a general overview, we report the minimum and maximum memory used in our experiments: on the smallest model coins, almost all algorithms need around 80 MB. Most memory is needed on the largest models AV and HW, where BVI, SI and HOP need between 1 GB and 1.5 GB, while QP can go up to 9.5 GB.

For a more detailed analysis, consider Table 5. The first row shows the number of times an algorithm needed less than twice as much memory as the algorithm who used the minimum memory. In other words, this is the number of times the algorithm performed "almost optimally" memory-wise. The second row shows the maximum relative overhead when compared to the minimum. For example, on one benchmark BVI needed 4 times as much as the minimum. Note that we omit optimizations in this table when they do not make a difference. For example, all 8 variants of SI perform the same in terms of memory.

We see that the memory usage of BVI, SI and HOP is very similar. HOP actually is close to the minimum memory on all benchmarks, while BVI and SI have a few where they need slightly more, namely BigMec_1e3, ManyMec_1e2 (both SI and BVI) and hm_30 (only BVI). Most QP variants perform worse, needing more memory several times. Moreover, this typically occurs on the large models AV and HW, where even the minimum memory is more than 1 GB, and thus a relative difference of 3 times larger is quite significant.

The topological variant of BVI is significantly worse on AV10_3 (8×) and BigMec_1e4 (29×). Interestingly, this problem does not occur for the topological variants of SI, QP or HOP. The original approach from [19] almost always needs more memory, often significantly so. The need for 88 times the memory occurs in the case study cdmsnMEC, with a difference of 120 MB to 10.5 GB. This highlights another[14] practical drawback of the transformation into normal form which blows up the game.

5.5.3. Theoretical comparison

To set the practical results into a more general context, we recall relevant theoretical properties of all three algorithm classes.

VI has two main drawbacks when compared to the other two: (i) it is an imprecise algorithm, i.e. it only converges in the limit and thus does not return the precise result in finite time. This can be addressed by introducing an additional rounding step after an exponential number of steps [10]; however, this is impractical, as the algorithm then *always* needs exponentially many steps. (ii) There are models where the exponential number of steps is necessary [5], so there is no hope to find a polynomial algorithm based on VI.

In contrast, SI and QP theoretically are precise methods, even though our practical implementation is not, since SI uses an imprecise, VI-based MDP solver and since QP exhibits numerical instability.

Both SI and QP still have the potential to yield a polynomial algorithm. For SI, this could be achieved by finding a polynomial bound for the number of iterations; currently, the best known bound is exponential. For QP, this could be achieved by finding a polynomially sized, convex QP (as solving convex QPs can be done in polynomial time [33]). Currently, our QP is exponential and it is unclear for both our and the previous polynomially sized QP [19] whether they are convex. If they are not, then the solving them is known to be NP-hard [42].

[14] Additionally to the one described in Section 4.1.4.

6. Conclusions

We have extended the three known classes of algorithm – value iteration, strategy iteration and quadratic programming – with several improvements and compared them both theoretically and practically.

In summary, for all algorithms, the structure of the underlying graph is more important than its size; thus knowledge about the model is relevant both for estimating the expected time, as well as the preferred algorithm and combination of optimizations. BVI and SI perform very similar on most models in our practical evaluation; each of them has some models where they are better. Quadratic/higher order programming is volatile and typically slower than the other two; however, the used solver has a huge impact, as we already see when changing between CPLEX, Gurobi and Minos. Thus, advances in the area of optimization problems could make this solution method the most practical.

In future work, we want to extend all algorithms to SGs with other objectives, e.g. total expected reward, mean payoff or parity. We expect the algorithms to work in a similar fashion as the ones presented for reachability in this paper, because for MDPs the other objectives are also similar to reachability. For example bounded value iteration in MDPs with total expected reward objective [4] differs mainly in the fact that one first has to compute an over-approximation, since it cannot start at 1 as the highest possible probability. Strategy iteration for MDPs with mean payoff [34] uses the same underlying idea as for reachability, but keeps track of two metrics (gain and bias) in order to decide how to improve a strategy. A key problem when extending bounded value iteration to total expected reward in SGs is that there are different semantics of the objective [15], and that in some cases the value is not the *least*, but the *greatest* fixpoint of the Bellman equations. Thus, value iteration from below can be stuck. To address this, probably a dual of the "deflating" [32] operation can be derived, which then increases the lower approximants in order to converge to the value.

A further direction for future work is coming up with a polynomially sized, convex QP. Using the normal form of [19], we have a polynomially sized QP. Possibly we could find a more practical polynomial QP by finding a solution for MECs that does not enumerate an exponential number of strategies. The key difficulty is in understanding whether the QP is convex or altering it so that it becomes convex. Overcoming this difficulty would give a positive answer to the long-standing open question whether simple stochastic games can be solved in polynomial time.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

This research was funded in part by the German Research Foundation (DFG) projects 383882557 Statistical Unbounded Verification (SUV) and 427755713 Group-By Objectives in Probabilistic Verification (GOPro).

References

[1] Pranav Ashok, Krishnendu Chatterjee, Jan Kretínský, Maximilian Weininger, Tobias Winkler, Approximating values of generalized-reachability stochastic games, in: LICS, ACM, 2020, pp. 102–115.
[2] Pranav Ashok, Jan Kretínský, Maximilian Weininger, PAC statistical model checking for Markov decision processes and stochastic games, in: CAV (1), in: Lecture Notes in Computer Science, vol. 11561, Springer, 2019, pp. 497–519.
[3] Christel Baier, Joost-Pieter Katoen, Principles of Model Checking, MIT Press, 2008.
[4] Christel Baier, Joachim Klein, Linda Leuschner, David Parker, Sascha Wunderlich, Ensuring the reliability of your model checker: interval iteration for Markov decision processes, in: CAV (1), in: Lecture Notes in Computer Science, vol. 1042, Springer, 2017, pp. 160–180.
[5] Nikhil Balaji, Stefan Kiefer, Petr Novotný, Guillermo A. Pérez, Mahsa Shirmohammadi, On the complexity of value iteration, in: ICALP, in: LIPIcs, vol. 132, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 102.
[6] Tomás Brázdil, Krishnendu Chatterjee, Martin Chmelik, Vojtech Forejt, Jan Kretínský, Marta Z. Kwiatkowska, David Parker, Mateusz Ujma, Verification of Markov decision processes using learning algorithms, in: ATVA, in: Lecture Notes in Computer Science, vol. 8837, Springer, 2014, pp. 98–114.
[7] Krishnendu Chatterjee, Luca de Alfaro, Thomas A. Henzinger, Strategy improvement for concurrent reachability and turn-based stochastic safety games, J. Comput. Syst. Sci. 79 (5) (2013) 640–657, https://doi.org/10.1016/j.jcss.2012.12.001.
[8] Krishnendu Chatterjee, Nathanaël Fijalkow, A reduction from parity games to simple stochastic games, in: GandALF, 2011, pp. 74–86.
[9] Krishnendu Chatterjee, Kristoffer Arnsfelt Hansen, Rasmus Ibsen-Jensen, Strategy complexity of concurrent safety games, in: MFCS, LIPIcs 83, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 55.
[10] Krishnendu Chatterjee, Thomas A. Henzinger, Value iteration, in: 25 Years of Model Checking, in: Lecture Notes in Computer Science, vol. 5000, Springer, 2008, pp. 107–138.
[11] Krishnendu Chatterjee, Thomas A. Henzinger, A survey of stochastic ω-regular games, J. Comput. Syst. Sci. 78 (2) (2012) 394–413, https://doi.org/10.1016/j.jcss.2011.05.002.
[12] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, Arjun Radhakrishna, Gist: a solver for probabilistic games, in: CAV, in: Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 665–669.
[13] Krishnendu Chatterjee, Joost-Pieter Katoen, Maximilian Weininger, Tobias Winkler, Stochastic games with lexicographic reachability-safety objectives, in: CAV (2), in: Lecture Notes in Computer Science, vol. 12225, Springer, 2020, pp. 398–420.
[14] Krishnendu Chatterjee, Koushik Sen, Thomas A. Henzinger, Model-checking omega-regular properties of interval Markov chains, in: FoSSaCS, in: Lecture Notes in Computer Science, vol. 4962, Springer, 2008, pp. 302–317.
[15] Taolue Chen, Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, Aistis Simaitis, Automatic verification of competitive stochastic systems, Form. Methods Syst. Des. 43 (1) (2013) 61–92, https://doi.org/10.1007/s10703-013-0183-7.
[16] Taolue Chen, Vojtech Forejt, Marta Z. Kwiatkowska, Aistis Simaitis, Clemens Wiltsche, On stochastic games with multiple objectives, in: MFCS, in: Lecture Notes in Computer Science, vol. 8087, Springer, 2013, pp. 266–277.
[17] Chih-Hong Cheng, Alois Knoll, Michael Luttenberger, Christian Buckl, GAVS+: an open platform for the research of algorithmic game solving, in: TACAS, in: Lecture Notes in Computer Science, vol. 6605, Springer, 2011, pp. 258–261.

[18] Anne Condon, The complexity of stochastic games, Inf. Comput. 96 (2) (1992) 203–224, https://doi.org/10.1016/0890-5401(92)90048-K.
[19] Anne Condon, On algorithms for simple stochastic games, in: Advances in Computational Complexity Theory, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 13, DIMACS/AMS, 1993, pp. 51–71.
[20] Costas Courcoubetis, Mihalis Yannakakis, The complexity of probabilistic verification, J. ACM 42 (4) (1995) 857–907, https://doi.org/10.1145/210332.210339.
[21] Decheng Dai, Rong Ge, Another sub-exponential algorithm for the simple stochastic game, Algorithmica 61 (4) (2011) 1092–1104, https://doi.org/10.1007/s00453-010-9413-1.
[22] Peng Dai Mausam, Daniel S. Weld, Judy Goldsmith, Topological value iteration algorithms, J. Artif. Intell. Res. 42 (2011) 181–209, Available at, http://jair.org/papers/paper3390.html.
[23] J. Filar, K. Vrieze, Competitive Markov Decision Processes, Springer-Verlag, 1997.
[24] Hugo Gimbert, Florian Horn, Simple stochastic games with few random vertices are easy to solve, in: FoSSaCS, in: Lecture Notes in Computer Science, vol. 4962, Springer, 2008, pp. 5–19.
[25] Serge Haddad, Benjamin Monmege, Interval iteration algorithm for MDPs and IMDPs, Theor. Comput. Sci. 735 (2018) 111–131, https://doi.org/10.1016/j.tcs.2016.12.003.
[26] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Kretínský, David Parker, Tim Quatmann, Enno Ruijters, Marcel Steinmetz, The 2019 comparison of tools for the analysis of quantitative formal models - (QComp 2019 competition report), in: TACAS (3), in: Lecture Notes in Computer Science, vol. 11429, Springer, 2019, pp. 69–92.
[27] Kristoffer Arnsfelt Hansen, Rasmus Ibsen-Jensen, Peter Bro Miltersen, The complexity of solving reachability games using value and strategy iteration, Theory Comput. Syst. 55 (2) (2014) 380–403, https://doi.org/10.1007/s00224-013-9524-6.
[28] Arnd Hartmanns, Benjamin Lucien Kaminski, Optimistic value iteration, in: CAV (2), in: Lecture Notes in Computer Science, vol. 12225, Springer, 2020, pp. 488–511.
[29] A.J. Hoffman, R.M. Karp, On nonterminating stochastic games, Manag. Sci. 12 (5) (1966) 359–370, https://doi.org/10.1287/mnsc.12.5.359.
[30] Rasmus Ibsen-Jensen, Peter Bro Miltersen, Solving simple stochastic games with few coin toss positions, in: ESA, in: Lecture Notes in Computer Science, vol. 7501, Springer, 2012, pp. 636–647.
[31] Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman, David Parker, A game-based abstraction-refinement framework for Markov decision processes, Form. Methods Syst. Des. 36 (3) (2010) 246–280, https://doi.org/10.1007/s10703-010-0097-6.
[32] Edon Kelmendi, Julia Krämer, Jan Kretínský, Maximilian Weininger, Value iteration for simple stochastic games: stopping criterion and learning algorithm, in: CAV (1), in: Lecture Notes in Computer Science, vol. 10981, Springer, 2018, pp. 623–642.
[33] Mikhail K. Kozlov, Sergei P. Tarasov, Leonid G. Khachiyan, The polynomial solvability of convex quadratic programming, USSR Comput. Math. Math. Phys. 20 (5) (1980) 223–228, https://doi.org/10.1016/0041-5553(80)90098-1.
[34] Jan Kretínský, Tobias Meggendorfer, Efficient strategy iteration for mean payoff in Markov decision processes, in: ATVA, in: Lecture Notes in Computer Science, vol. 10482, Springer, 2017, pp. 380–399.
[35] Jan Kretínský, Tobias Meggendorfer, Of cores: a partial-exploration framework for Markov decision processes, in: CONCUR, in: LIPIcs, vol. 140, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 5.
[36] Marta Kwiatkowska, Gethin Norman, David Parker, Gabriel Santos, PRISM-games 3.0: stochastic game verification with concurrency, equilibria and time, in: CAV (2), in: Lecture Notes in Computer Science, vol. 12225, Springer, 2020, pp. 475–487.
[37] Jan Křetínský, Emanuel Ramneantu, Alexander Slivinskiy, Maximilian Weininger, Comparison of algorithms for simple stochastic games, Electr. Proc. Theor. Comput. Sci. 326 (2020) 131–148, https://doi.org/10.4204/eptcs.326.9.
[38] Ludwig Walter, A subexponential randomized algorithm for the simple stochastic game problem, Inf. Comput. 117 (1) (1995) 151–155, https://doi.org/10.1006/inco.1995.1035.
[39] Kittiphon Phalakarn, Toru Takisaka, Thomas Haas, Ichiro Hasuo, Widest paths and global propagation in bounded value iteration for stochastic games, in: CAV (2), in: Lecture Notes in Computer Science, vol. 12225, Springer, 2020, pp. 349–371.
[40] Martin L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, Wiley Series in Probability and Statistics, Wiley, 1994.
[41] Tim Quatmann, Joost-Pieter Katoen, Sound value iteration, in: CAV (1), in: Lecture Notes in Computer Science, vol. 10981, Springer, 2018, pp. 643–661.
[42] Sartaj Sahni, Computationally related problems, SIAM J. Comput. 3 (4) (1974) 262–279, https://doi.org/10.1137/0203021.
[43] Rafal Somla, New algorithms for solving simple stochastic games, Electron. Notes Theor. Comput. Sci. 119 (1) (2005) 51–65, https://doi.org/10.1016/j.entcs.2004.07.008.
[44] María Svorenová, Marta Kwiatkowska, Quantitative verification and strategy synthesis for stochastic games, Eur. J. Control 30 (2016) 15–30, https://doi.org/10.1016/j.ejcon.2016.04.009.
[45] Maximilian Weininger, Tobias Meggendorfer, Jan Kretínský, Satisfiability bounds for ω-regular properties in bounded-parameter Markov decision processes, in: CDC, IEEE, 2019, pp. 2284–2291.
[46] Uri Zwick, Mike Paterson, The complexity of mean payoff games on graphs, Theor. Comput. Sci. 158 (1,2) (1996) 343–359, https://doi.org/10.1016/0304-3975(95)00188-3.