DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Facilitating Experimental Analysis of Algorithms for Stochastic Games: Random Model Generation and Mining the Results

Alexander Slivinskiy

DEPARTMENT OF INFORMATICS

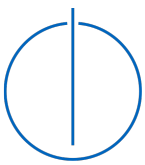
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Facilitating Experimental Analysis of Algorithms for Stochastic Games: Random Model Generation and Mining the Results

Unterstützen der experimentellen Analyse von Algorithmen für das Lösen stochastischer Spiele: Erzeugung von Zufallsmodellen und Auswertung der Ergebnisse

Author:	Alexander Slivinskiy
Supervisor:	Prof. Dr. Jan Křetínský
Advisor:	Maximilian Weininger, Muqsit Azeem
Submission Date:	



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Alexander Slivinskiy

Acknowledgments

- I want to thank Prof. Dr. Jan Křetínský for giving me the opportunity to work on this topic. Working on this task was a very valuable experience for me.
- I want to thank Maximilian Weiniger and Muqsit Azeem for always being encouraging regarding the direction this thesis is taking. I want to thank them for always being very patient and helpful with all the questions and problems I came across throughout the thesis.
- I want to thank Calvin Chau for the endless conversations about theoretical computation and for sparking new ideas when I felt like reaching a dead-end in any task.

Abstract

Simple stochastic games are a formalism to model and verify reactive systems in stochastic settings. We consider reachability games, where the goal of the first player is to reach a set of targets. The goal of the second player is to prevent the first player from reaching any target. By now, there are many algorithms to solve stochastic games with reachability objectives. However, there is a lack of experimental data and insight about which circumstances favor which algorithm, leaving users without a solid baseline for an informed decision on which algorithm to use to solve their problem. In this thesis, we generate reachability games randomly to expand the benchmarking data set. Furthermore, we provide tools for algorithm analysis that scale with growing numbers of both the number of stochastic games we solve and the number of algorithms we analyze. Lastly, we draw conclusions about the performance of algorithms solving simple stochastic games in relation to the structural properties of the stochastic game by utilizing the models and tools we introduce.

Contents

1. Introduction

We live in a time in which technology and computer systems are all around us and are a crucial part of our daily life. Our dependency on these systems is increasing. No matter if we consider sending an email, paying online with our smartphone, being driven by an autonomously driving car, or a factory where machines have to perform tasks: we as consumers and also as developers want to be sure that the systems and the programs are doing exactly what they should.

Verification is a field of computer science focused on providing formal guarantees for systems. One widely used approach to do so is *model checking* where a real system is simplified to a theoretical model. The model is then used to assess which properties about the given system hold.

When modeling a real-world system, occasional events like bad weather while driving or that a part breaking in the factory may be necessary to take into account. Thus, we need to quantify how often these realistic events occur and make our theoretical model probabilistic. Through *probabilistic model checking*, it is possible to provide a guarantee with a probability that the model is going to behave as it should. One prominent way to express various probabilistic systems is to model them as *simple stochastic games (SGs)*, which are two-player zero-sum games that are played on graphs. The vertices of the graph belong to either one player or the other, and in every state, there is a set of actions that lead in a probabilistic manner into other states. One player tries to achieve his goal, and the other tries to prevent him from this. This is a natural way to model, for example, a system in an unknown environment. The goal we consider is reachability, i.e. the first player has to reach a certain set of states to win, while the other player has to prevent this. We call these types of games *reachability games*.

A practically relevant problem is to compute the probability of the first player reaching one of these goal states. This way we derive what we want to verify in the first place: the probabilistic guarantee that the real-world system behaves as we expect it to.

The three common solution techniques to solve reachability problems for stochastic games are quadratic programming, strategy iteration (also known as policy iteration), and value iteration. In practice, value iteration is considered the fastest of the three solution techniques. However, there is an adversarial example proving that value iteration may need exponentially many steps to solve a problem [Bal+19].

For all the three techniques for solving reachability games, there are many optimizations and changes available that affect their performance. However, at the moment, there is a lack of experimental data that enables a solid quantitative assertion of the performance of the solution techniques and their extensions. In the PRISM-games benchmark suite, there are 10 distinct

models. With the models "Hallway" and "Avoid the Observer" [Cha+20] we are aware of about a dozen real-world case studies. Since the performance of every algorithm depends on the underlying structural properties of the stochastic game at hand [Kře+20], an algorithm that performs well on the available case studies could still overall be an unfavorable choice since the dataset might not contain enough structural variance to enable an adequate evaluation of the algorithm performance.

Our contribution

To tackle these problems, we provide the following contributions:

- Extend the set of stochastic games by generating models randomly.
- Introduce analysis tools that are fit for a growing number of stochastic games and solution techniques.
- Analyze the performance of the algorithms for stochastic games. In particular, we analyze the impact of structural properties of models on the algorithms, structural biases of the real case studies, and algorithm performance.

We see our contributions as the first steps towards making stochastic games a practically relevant solution for modeling systems. The more insights we gather about the relation between structural properties and algorithms solving stochastic games, the more sophisticated solutions can be found for problems that current prominent algorithms have.

Note that we will only focus on value iteration, strategy iteration, and their extensions since it was already shown in [Kře+20] that at the moment quadratic programming is not a recommended approach to tackle reachability problems in stochastic games. When testing quadratic programming on randomly generated models, we confirmed these results.

The rest of the thesis is structured as follows: Chapter 2 introduces the necessary preliminaries for this thesis. In Chapter 3, we describe the extensions we have implemented in the PRISM model checker for strategy iteration and value iteration. First, Chapter 4 provides the benefits as well as the theoretical aspects to the random generation of stochastic games, then Chapter 5 presents implementation details and a manual on how to use our implementation. In Chapter 6, we describe the tools we implemented for the analysis of both stochastic games and solution techniques. Chapter 7 presents benchmarks of the algorithms on the reachability games, as well as results of the model and algorithm analysis. Lastly, Chapter 8 concludes our work.

Related work

Markov Decision Processes are a generalization of Markov Chains [Put14] [GS06, Ch. 11].

The first to introduce the concept of stochastic games as well as value iteration as solution algorithm was Shapley in [Sha53]. Condon has shown that the complexity of solving simple stochastic games is in **NP** ∩ co-**NP** [Con92]. Condon has also introduced both quadratic programming and strategy iteration as algorithms for stochastic reachability games in [Con93].

Both value iteration and strategy iteration are also solution methods for MDPs [Put14][HK66]. However, the problem with standard value iteration in both MDPs and SGs is that it could be arbitrary imprecise [HM18] and could in practice run for exponentially many steps [Bal+19]. Recently, several heuristics were introduced that provide a guarantee on how close value iteration is away from the result for MDPs [HM18] and SGs [Kel+18]. [Pha+20] provided for these heuristics for SGs a new way of solving structures called end components.

[Kře+20] has recently shown that at the moment quadratic programming is not competitive with value iteration and strategy iteration, so we will not consider it in this thesis. Furthermore, it has introduced optimizations for both strategy iteration and value iteration for stochastic games that solve via divide and conquer by first providing a topological partitioning of the stochastic game.

The tools implementing the standard ways of standard iteration and/or value iteration algorithms for SGs are PRISM-games [Che+13], GAVS+ [Che+11b] and GIST [Cha+10]. However, GAVS+ is not maintained anymore, and exists more for educational purposes. GIST considers ω-regular objectives but performs only qualitative analysis. For MDPs (games with a single player), the recent friendly competition QComp [Hah+19] gives an overview of the existing tools.

2. Preliminaries

In this chapter, we introduce the underlying definitions necessary to have an understanding of the notation and foundation on which this thesis is built on. To get an overview of what simple stochastic games are and how they are played, we provide an intuition in Section 2.1. In Section 2.2, we introduce the formal definition of simple stochastic games, which is the stochastic model we consider. We formalize the notion of players having strategies in Section 2.3 and define the semantics of simple stochastic games in Section 2.4. We then consider special subparts of the state space of stochastic games called end components in Section 2.5. Next, we recall an important graph property known as strongly connectedness in Section 2.6. Lastly, we provide short descriptions of the most prominent algorithms that solve simple stochastic games and some optimizations to them in Section 2.7.

2.1. Intuition

A *simple stochastic game* is a two-player game introduced by Shapley in [Sha53] played on a graph G whose vertices S are partitioned into the two sets $\mathsf{S}_{\Box}$ and $\mathsf{S}_{\bigcirc}$. Each set belongs to a player. We call the players Maximizer and Minimizer. We refer to vertices also as states.

At the beginning of the game, a token is placed on the initial vertex s_0. The Maximizer's goal is to move the token to any *target* $\mathsf{f} \in F \subseteq \mathsf{S}$, while the Minimizer's goal is that the token never reaches any target-vertex f. Stochastic games with such objectives are called *reachability games*. To move the token, every vertex $s \in S$ has a finite set of actions $\mathsf{Av}(\mathsf{s})$ that can be taken if the token is on this vertex. Each action causes the transition of the token to another vertex with a probability according to the probability distribution $\delta : S \to [0,1] \subset \mathbb{Q}$. If the token is in a Maximizer-state $s \in \mathsf{S}_{\Box}$, then the Maximizer decides which action to pick. If the token is in a Minimizer-state $\mathsf{s} \in \mathsf{S}_{\bigcirc}$, the Minimizer decides. Figure 2.1 provides an example of a simple stochastic game.

The problem we consider in this thesis is: *Given a token in the initial state* s_0*, what is the probability that the token reaches any target state* $\mathsf{f} \in F$ *if both players play optimally?*

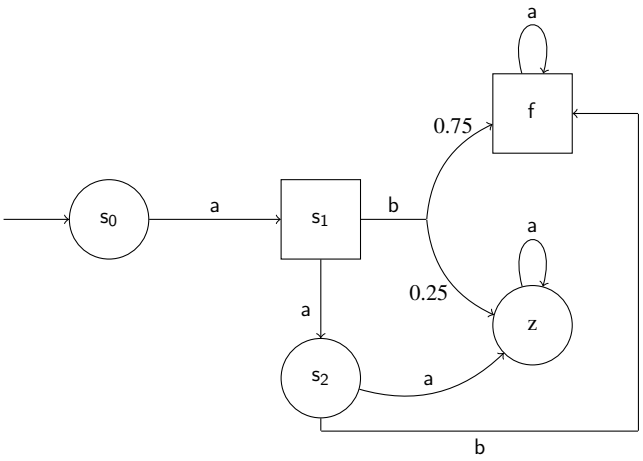


Figure 2.1.: An example of a simple stochastic game. $s_0, s_1, s_2, \mathsf{f}, \mathsf{z}$ are states. a, b are actions. s_1 and f are Maximizer-states, while the rest of the states belongs to the Minimizer. f is a target and z is a so-called sink, i.e. a state that has no path to the target. 0.75 and 0.25 are the transition probabilities that a token in s_1 moved through action b would either be moved to f or z

2.2. Stochastic games

We define now simple stochastic games, also referred to as stochastic games [Con92]. To do so, we need to introduce *probability distributions*. 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 $\mathcal{D}(X)$.

Definition 2.2.1 (SG)**.** *A* stochastic game (SG) *is a tuple* $(\mathsf{S}, \mathsf{S}_\Box, \mathsf{S}_\bigcirc, \mathsf{s}_0, \mathsf{A}, \mathsf{Av}, \delta)$ *where* S *is a finite set of* states *partitioned*[1] *into the sets* $\mathsf{S}_\Box$ *and* $\mathsf{S}_\bigcirc$ *of states of the player* Maximizer *and* Minimizer *respectively,* $\mathsf{s}_0 \in \mathsf{S}$ *is the* initial *state,* A *is a finite set of* actions*,* $\mathsf{Av} : \mathsf{S} \to 2^{\mathsf{A}}$ *assigns to every state a set of* available *actions, and* $\delta : \mathsf{S} \times \mathsf{A} \to \mathcal{D}(\mathsf{S})$ *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.*

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

We use a model of stochastic games similar to [Kel+18].

The set of successors that can be reached from a state s taking the action $\mathsf{a} \in \mathsf{Av}(\mathsf{s})$ is described by $\mathsf{Post}(\mathsf{s},\mathsf{a}) := \{\mathsf{s}' \mid \delta(\mathsf{s},\mathsf{a},\mathsf{s}') > 0\}$. We assume that every state has at least one action it can take so that for all $\mathsf{s} \in \mathsf{S}$ it holds that $\mathsf{Av}(\mathsf{s}) \neq \emptyset$. Furthermore, we call an action deterministic if $|\mathsf{Post}(\mathsf{s},\mathsf{a})| = 1$.

[1] I.e., $\mathsf{S}_\Box \subseteq \mathsf{S}$, $\mathsf{S}_\bigcirc \subseteq \mathsf{S}$, $\mathsf{S}_\Box \cup \mathsf{S}_\bigcirc = \mathsf{S}$, and $\mathsf{S}_\Box \cap \mathsf{S}_\bigcirc = \emptyset$.

2.3. Strategies

As mentioned in Section 2.1, the Maximizer selects the action for states $\mathsf{s} \in \mathsf{S}_\Box$, and the Minimizer chooses the action for states $\mathsf{s} \in \mathsf{S}_\bigcirc$. This is formalized as *strategies* with $\sigma : \mathsf{S}_\Box \to \mathcal{D}(\mathsf{A})$ being a Maximizer strategy and $\tau : \mathsf{S}_\bigcirc \to \mathcal{D}(\mathsf{A})$ being a Minimizer strategy, such that $\sigma(\mathsf{s}) \in \mathcal{D}(\mathsf{Av}(\mathsf{s}))$ for all $\mathsf{s} \in \mathsf{S}_\Box$ and $\tau(\mathsf{s}) \in \mathcal{D}(\mathsf{Av}(\mathsf{s}))$ for all $\mathsf{s} \in \mathsf{S}_\bigcirc$.

Since we deal with reachability games, we will consider only *memoryless positional strategies*. This means that for each state there is exactly one fixed action that is taken: $\forall \mathsf{s} \in \mathsf{S}_\Box : \exists \mathsf{a} \in \mathsf{Av}(\mathsf{s}) : \sigma(\mathsf{s},\mathsf{a}) = 1$ and $\forall \mathsf{s} \in \mathsf{S}_\bigcirc : \exists \mathsf{a} \in \mathsf{Av}(\mathsf{s}) : \tau(\mathsf{s},\mathsf{a}) = 1$. For reachability games, these types of strategies are optimal [Con92].

2.4. Reachability objective

We have introduced now what a SG is and how to fix strategies, but it is still unclear what the objectives of the players are.

As mentioned in Section 2.1, at the beginning of the game a token is placed on the initial vertex s_0. Additionally, a subset $F \subseteq \mathsf{S}$ is provided as input. The Maximizer's goal is to maximize the probability of reaching any *target* $\mathsf{f} \in F$ while the Minimizer's goal is to minimize the probability that the token reaches any target f. Vertices from which the Maximizer can never reach any target are referred to as *sinks* $\mathsf{z} \in Z \subseteq \mathsf{S}$. Thus, once the token reaches any sink z the Maximizer loses the game.

Since the objective of the Maximizer is to reach a target state f, we call this stochastic game a *reachability game*. In these kinds of games, we do not care what happens after reaching any target or sink state. Therefore, we assume for simplicity that each target f and each sink z has only one action which is a self-loop with transition probability 1. Figure 2.1 provides an example of a SG with one target f and one sink z.

Intuitively, we measure how good a strategy σ for s is by the probability that the token would get from s to any target f if the Minimizer is using strategy τ. This is what we call the *value* V *of* $\mathsf{s} \in \mathsf{S}$ *using* (σ,τ):

$$\mathsf{V}_{\sigma,\tau}(\mathsf{s}) = \sum_{f \in F} p_s^{\sigma,\tau}(f)$$

where $p_s^{\sigma,\tau}(f)$ is the probability that s reaches f in $\mathsf{G}^{\sigma,\tau}$ in arbitrary many steps. Note that this is a high-level definition that is based on unique probability distributions over measurable sets of infinite paths that are induced by fixing strategies [BK08, Ch. 10]. However, we do not need this level of detail for this thesis and will omit it for easier readability. We define $\mathsf{V}(\mathsf{s},\mathsf{a}) = \sum_{\mathsf{s}' \in \mathsf{Post}(\mathsf{s},\mathsf{a})} \delta(\mathsf{s},\mathsf{a},\mathsf{s}')\mathsf{V}(\mathsf{s}')$.

As we expect that both Maximizer and Minimizer play as good as possible and pick the optimal strategies for their goal, usually one is only interested in the value of states using these optimal

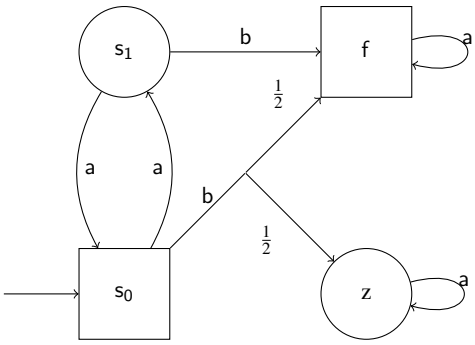


Figure 2.2.: An example of a non-stopping SG with an end component of size 2. If in s_0 action a and in s_1 action a are chosen for some strategies σ, τ the game does never stop. $\mathsf{V}_{\sigma,\tau}(s_0) = 0$ because the probability to reach any $f \in F$ is 0 with these strategies.

strategies. We define this as the *(optimal) value* of $\mathsf{s} \in \mathsf{S}$.

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

The second equality is due to [Con92]. Again, this is a simplified definition derived from using the supremum and infimum instead of the maximum and minimum. However, as we are dealing with memoryless positional strategies, a maximum and a minimum always exist [BK08, Ch. 10].

We define the *value of a SG* by the probability that the Maximizer wins starting in the initial state s_0. We are mainly interested in $\mathsf{V}(\mathsf{s}_0)$ throughout this thesis. Condon has shown that the complexity of solving the corresponding decision problem is in **NP** $\cap$ co-**NP** [Con92].

For example, in Figure 2.1 the value of the initial state s_0 and therefore of the SG is 0.75. This is because s_0 has to pick action a leading into s_1. For the Maximizer, action b is better than action a in s_1. Action a in state s_1 leads into s_2, which in turn would always pick action a leading always into the z. However, action b for state s_1 has a 75% chance of reaching the target, thus being the better action for the Maximizer.

2.5. End components

Some stochastic games contain subsets $T \subseteq \mathsf{S}$, for which the players can choose strategies such that the token remains forever in one of these subsets and therefore never reaches any target or sink state. We call such a subset an *end component (EC)*. Figure 2.2 illustrates a simple SG with an end component. If both s_0 and s_1 pick action a the token would never reach neither target f nor sink z.

For the definition, we need to introduce *finite paths*. A finite path ρ is a finite sequence $\rho = \mathsf{s}_0\mathsf{a}_0\mathsf{s}_1\mathsf{a}_1 \ldots \mathsf{s}_k \in (\mathsf{S} \times \mathsf{A})^* \times \mathsf{S}$, such that for every $i \in [k-1]$, $\mathsf{a}_i \in \mathsf{Av}(\mathsf{s}_i)$ and $\mathsf{s}_{i+1} \in \mathsf{Post}(\mathsf{s}_i, \mathsf{a}_i)$.

2

Definition 2.5.1 (End component (EC))**.** *[Kel+18] A non-empty set $T \subseteq \mathsf{S}$ of states is an* end component (EC) *if there is a non-empty set $B \subseteq \bigcup_{\mathsf{s} \in T} \mathsf{Av}(s)$ of actions such that*

1. *for each $\mathsf{s} \in T, \mathsf{a} \in B \cap \mathsf{Av}(\mathsf{s})$ we do* not *have (s, a) leaves T,*
2. *for each $\mathsf{s}, \mathsf{s}' \in T$ there is a finite path $\mathsf{w} = \mathsf{s}\mathsf{a}_0 \dots \mathsf{a}_n \mathsf{s}' \in (T \times B)^* \times T$, 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 $T \subseteq T'$. Note that sinks and targets are maximal end components of size 1. Given a SG G, the set of its MECs is denoted by $\mathsf{MEC}(\mathsf{G})$ and can be computed in polynomial time [CY95].

Computing the value of states that are part of an end component that is not a sink is for many algorithms a non-trivial task that requires additional concepts.

2.6. Strongly connected components

Another important structural concept we reason about is a strongly connected component (SCC)

Definition 2.6.1 (Strongly Connected Component (SCC))**.** *A set of states $V \subseteq \mathsf{S}$ is* strongly connected *iff for every pair of states $(\mathsf{s}, \mathsf{s}') \in (V \times V)$ there is a path from s to s' and a path from s' to s. V is a* strongly connected component *iff V is strongly connected and maximal, i.e. there is no strongly connected set $W \subseteq \mathsf{S}$ such that $V \subset W$.*

Since every state belongs to exactly one strongly connected component, the set of strongly connected components partitions the set of states S. Decomposing S into its strongly connected components can be useful because it is possible to induce subgames of any stochastic game based on its SCCs and solve the SG with a divide-and-conquer approach. Instead of computing the values of the states in the stochastic game, one instead computes the value of the states in the subgames. To use this approach, we need to partition S into its SCCs, and we need to compute a topological enumeration of the SCCs to know in which order to process the components. Tarjan's algorithm for strongly connected components [Tar72] does both. In this thesis, we refer to algorithms that use SCC decomposition to solve stochastic games as topological algorithms.

2.7. Algorithms for stochastic games

Next, we recall some of the most prominent algorithms that *solve* stochastic games, i.e. compute the value of any stochastic game.

The three most common approaches to solving stochastic games are *value iteration*, *strategy iteration* and *quadratic programming*.

[2] $[l] = 1, 2, \dots, l$, where $l \in \mathbb{N}$

2.7.1. Value iteration

To compute the value function V for an SG, the following partitioning of the state space is useful: firstly the goal states T that surely reach any target state $\mathsf{f} \in F$, secondly the set of *sink states* Z that do not have a path to the target and finally the remaining states $\mathsf{S}^?$. For T and Z (which can be easily identified by graph-search algorithms), the value is trivially 1 respectively 0. Thus, the computation only has to focus on $\mathsf{S}^?$.

The well-known approach of value iteration (**VI**) leverages the fact that V is the least fixpoint of the *Bellman equations*, cf. [CH08]:

$$\mathsf{V}(s) = \begin{cases} 1 & \text{if } s \in \mathsf{T} \\ 0 & \text{if } s \in Z \\ \max_{a \in \mathsf{Av}(s)} \left(\sum_{s' \in \mathsf{S}} \delta(s,a,s') \cdot \mathsf{V}(s') \right) & \text{if } s \in \mathsf{S}^?_\Box \\ \min_{a \in \mathsf{Av}(s)} \left(\sum_{s' \in \mathsf{S}} \delta(s,a,s') \cdot \mathsf{V}(s') \right) & \text{if } s \in \mathsf{S}^?_\bigcirc \end{cases} \tag{2.1}$$

Now we define[3] the Bellman operator $\mathcal{B} : (\mathsf{S} \to \mathbb{Q}) \to (\mathsf{S} \to \mathbb{Q})$:

$$\mathcal{B}(\mathsf{f})(\mathsf{s}) = \begin{cases} \max_{a \in \mathsf{Av}(s)} \left(\sum_{s' \in \mathsf{S}} \delta(s,a,s') \cdot f(s') \right) & \text{if } s \in \mathsf{S}_\Box \\ \min_{a \in \mathsf{Av}(s)} \left(\sum_{s' \in \mathsf{S}} \delta(s,a,s') \cdot f(s') \right) & \text{if } s \in \mathsf{S}_\bigcirc \end{cases} \tag{2.2}$$

Value iteration starts with the under-approximation

$$\mathsf{L}_0(s) = \begin{cases} 1 & \text{if } s \in \mathsf{T} \\ 0 & \text{otherwise} \end{cases}$$

and repeatedly applies the Bellman operator. Since the value is the least fixpoint of the Bellman equations and $\mathsf{L}_0 \leq \mathsf{V}$ is lower than the value, this converges to the value in the limit [CH08] (formally $\lim_{i \to \infty} \mathcal{B}^i(\mathsf{L}_0) = \mathsf{V}$).

Bounded value iteration

While standard value iteration is often fast in practice, it has the drawback that it is not possible to know the current difference between $\mathcal{B}^i(\mathsf{L}_0)$ and V for any given i. To address this, one can employ *bounded value iteration* (**BVI**, also known as interval iteration [HM18; Brá+14; Kel+18]) It additionally starts from an over-approximation U_0, with $\mathsf{U}_0(s) = 1$ for all $s \in \mathsf{S}$. However, applying the Bellman operator to this upper estimate might not converge to the value, but some

[3] In the definition of $\mathcal{B}$, we omit the technical detail that for goal states $s \in \mathsf{T}$, the value has to remain 1. Equivalently, one can assume that all goal states are absorbing, i.e. only have self looping actions.

greater fixpoint instead due to end components. See [Kře+20, Example 2] for an example. To fix this issue, **BVI** computes which states outsides of the end component the Maximizer can reach from within the end component if Minimizer plays optimally. The upper bound for these states inside the end component is then be defined by the upper bound of the states outside the end component. This approach is referred to as *deflating* [Kel+18]. With deflating, the upper bound decreases monotonically towards the least fixpoint. Once $\forall \mathsf{s} \in \mathsf{S} : \mathsf{U}(s) - \mathsf{L}(s) \leq \varepsilon$, **BVI** terminates and reports $\mathsf{V}(\mathsf{s}_0) = \frac{\mathsf{U}_0 + \mathsf{L}_0}{2}$. In addition to **BVI**, we will consider the widest path optimization **WP** introduced in [Pha+20] that offers an alternative to deflating for the correct handling of end components.

Optimistic value iteration

The heuristic *optimistic value iteration* (**OVI**) takes a similar approach to bounded value iteration, but instead of calculating the upper bound at all times along with the lower bound, an upper bound B is guessed from the lower bound by adding a small ε to the lower bound. The bound is guessed when no states value changes by more than $\varepsilon' \leq \varepsilon$. After the guessing, the verification phase starts, in which **OVI** tries to verify that the guessed bound is truly an upper bound. During this phase, the algorithm only iterates on the guessed bound. There are three options of how the verification phase terminates:

1. The guessed bound is verified to be an upper bound, i.e. $\forall \mathsf{s} \in \mathsf{S} : \mathcal{B}(B)(\mathsf{s}) \leq B(\mathsf{s})$. In this case, **OVI** terminates since it has computed the values of the state up to ε-precision.

2. The guessed bound is a tighter lower bound of V than the current one. In this case, the guessed bound is used a as new lower bound and the verification phase is delayed to the next time no states value changes by more than ε'.

3. The guessed bound is not identified as either lower or upper bound. In this case, verification fails, **OVI** aborts the verification phase, halves ε' and continues iterating on the lower bound until the next verification phase may begin.

Optimistic value iteration was introduced in [HK19] for MDPs and extended to SGs in [Aze+ed].

There are more extensions to value iteration. One of them is the learning-based approach introduced in [Brá+14]. However, it was already benchmarked in [Kře+20], and its random nature is fundamentally different to the solution approaches we analyze in this thesis which is why we omit it.

In practice, value iteration and its extensions are believed to be the fastest approaches to solve stochastic games. However, as shown in [Bal+19], there are counterexamples where value iteration requires exponentially many steps to converge and its performance is worse than the other solution approaches like strategy iteration.

2.7.2. Strategy iteration

In strategy iteration [HK66][Con93], instead of computing value-vectors we compute a sequence of strategies and terminate when an optimal strategy for both players is reached. We start at an arbitrary strategy for the Maximizer and repeatedly compute the best response the Minimizer can make. Next, the Maximizer's strategy is then improved greedily. The computed sequence of Maximizer strategies is monotonically increasing regarding the value and converges to the optimal strategy [CdH13, Theorem3]. For games with end components, the initial strategy may not be completely arbitrary but has to ensure that at least one target or sink is reached almost surely. Finding such an initial strategy can be achieved with the attractor strategy [CdH13, Section 5.3].

Whenever a Maximizer strategy is fixed, the resulting SG is simplified to an MDP. From there on, any solution method for MDPs can be applied. We use three different variants of solving the MDPs: value iteration ($\mathbf{SI_{VI}}$), linear programming [Put14] ($\mathbf{SI_{LP}}$), or strategy iteration ($\mathbf{SI_{SI}}$). When applying $\mathbf{SI_{SI}}$, the Minimizer picks a strategy, and the MDP is reduced to a Markov chain. We express the Markov chain as a degenerate linear program where every constraint is an equation and hand it to a linear programming solver.

Since trying out every possible deterministic positional strategy guarantees that the optimal strategy is found, the trivial upper bound on the number of iterations for strategy iteration is exponential in the number of actions per state.

However, so far, no example confirmed that there is a stochastic game where strategy iteration truly needs exponentially many iterations.

2.7.3. Quadratic programming

In quadratic programming, the stochastic game is encoded into a mathematical framework called a quadratic program. The encoded problem is then handed to a solver, which uses algorithms specific to mathematical programming in general. While solving an arbitrary quadratic problem is known to be **NP**-complete, convex quadratic programs can be solved in polynomial time. At the moment, it is open whether it is possible to encode every stochastic game into a convex quadratic program. In practice, [Kře+20] has shown that quadratic programming is not competitive to value iteration and strategy iteration even if using state-of-the-art solvers. Thus, we will not include it in our benchmarks.

2.7.4. Optimizations

In addition to the algorithms and their extensions, there are various optimizations that we will consider in this thesis:

G: The Gauss-Seidel variant of value iteration for **BVI** and **OVI**. Bellman updates happen in-place and state by state. Thus, states updated later use the already updated values of previous

states.

D: The idea from [Kel+18] to only deflate every 100 steps for **BVI**. By deflating only every 100 steps, there is less time spent on searching for the end components that need to be deflated.

T: A topological decomposition of SG into its strongly connected components for **BVI**, **OVI**, and $\mathbf{SI_{LP}}$. The stochastic game is decomposed into its SCCs, then the values of the SCCs are computed along their topological order. This allows the algorithms to solve smaller sub-problems before instead of solving the whole SG at once.

$\mathbf{BVI^{PT}}$: $\mathbf{BVI^{PT}}$ as in *precise and topological* is an extension to value-iteration-based algorithms that builds on top of the idea of topological sorting of [Kře+20] that was introduced in [Aze+ed]. We analyze the graph of the underlying the stochastic game, detect the strongly connected components, and create a topological sorting based on the SCCs. To ensure that every SCC provides an exact result to the next one, we take the ε-precise values of the states of the finished SCC and infer all the strategies that may yield this value. For each state, we combine each action whose value is within the ε-bounds. Next, we fix the Maximizer and the Minimizers strategies separately, yielding two MDPs. We then perform one strategy iteration step on both MDPs. For the iteration, we recommend to the player which has to choose the action that combines all actions within the ε-bounds. If in both MDPs the strategies do not change, we confirmed that the actions we suggested truly are optimal.

Algorithm Performance and Stochastic Game Structure

As [HM18] and [Kře+20] have shown, the performance of the algorithms is very dependent on the structural properties of the stochastic game. Structural properties are, for example, the number of transitions an action has, the size of the biggest MEC, or the number of strongly connected components in a stochastic game. While some algorithms may struggle with solving certain stochastic games, others may solve them without problems. However, at the moment, there are only a few insights on which structural properties should be taken into account when deciding on which algorithm to take.

Thus, we analyze the impact of different structural properties on the performance of the algorithms that solve SG. To evaluate performance, we do not only use real case studies but also randomly generated stochastic games.

3. Implemented algorithm extensions

As part of the thesis, we have implemented the following extensions to value iteration and strategy iteration:

$\mathbf{BVI}^{\mathbf{PT}}$

To solve the MDPs in $\mathbf{BVI}^{\mathbf{PT}}$, we used strategy iteration. This yields two discrete-time Markov chains (DTMCs). We then solve the DTMCs by expressing them as linear programs and handing them to linear programming solvers.

Widest path for PRISM 3

Widest Path was introduced as an alternative to solving maximal end components for bounded value iteration [Pha+20]. We reimplemented it in version 3 of PRISM.

Linear programming for MDPs

Although linear programming is a well-known approach to solving MDPs, it is believed to perform worse than value iteration or strategy iteration and was not implemented in PRISM. We have implemented solving MDPs with linear programming as in [Put14]. This allows for a combination of strategy iteration to fix one player's strategy and linear programming to solve the induced MDP.

4. Randomly generated models

In this chapter, we provide an overview of the theory of randomly generating models for stochastic games. We discuss first the benefits of randomly generated models, explain which types of stochastic games we want to generate, and then present our algorithm for random model generation. Lastly, we present a more flexible way of creating handcrafted models.

4.1. Why to use randomly generated models

Using randomly generated problems is a classic approach to test algorithms. A key benefit of randomly generated models is that their structure can deviate from currently available case studies and thus express structural features that usually do not occur. This can lead to a difference in algorithm performance and may become relevant when new case studies emerge. Also, randomly generated models can be created both automatically and in large numbers.

To make a quantitative assertion about the performance of the algorithms we need first a broad spectrum of models we can test the algorithms on. At the moment, we are aware of 12 case studies. These case studies often have parameters that can be adjusted, resulting in infinitely many stochastic games. However, stochastic games derived from the same problem will have similar graph structures and thus still cover only a certain portion of all possible graph structures that may occur.

We need more distinct models to provide a solid algorithm comparison. Creating handcrafted reachability problems is useful to make comparisons on edge-case situations and show off the weaknesses and strengths of categories of algorithms. Examples for this are [HM18], which is an adversarial example for value iteration, the model 'BigMEC' [Kře+20], which is adversarial for quadratic programming, and 'MulMEC' [Kře+20], which was presented as a case that is especially easy for widest path bounded value iteration [Pha+20]. However, by nature handcrafted models cover only extreme cases and thus provide little test coverage for more general problems.

To tackle these issues, we introduce randomly generated models. This allows us to generate a lot of models in a short time span. Since the graph structure is generated randomly, the models can have arbitrary structural properties and thus enable broad test coverage for algorithm performance.

4.2. Constraining the random generation process

While it is possible to completely randomize every property of a model, this procedure would yield graphs with undesirable properties. We are only interested in models in which the initial state can reach every state of the underlying graph of a stochastic game. This is because all the states that can never be reached by the initial state do not influence the value of the game and thus add unnecessary complexity. We refer to the set of stochastic games where the initial state can reach every other state in the game by $\mathcal{G}_{Reachable}$.

A viable approach is to sample uniformly from $\mathcal{G}_{Reachable}$ to minimize the number of structural biases in our models. However, since $\mathcal{G}_{Reachable}$ is infinite, we would be required to discretize the transition probabilities and limit the number of states, restricting $\mathcal{G}_{Reachable}$. Instead, we use an iterative generation process. While not sampling $\mathcal{G}_{Reachable}$ space uniformly, it is easy to implement, modify, and can generate any stochastic game in $\mathcal{G}_{Reachable}$.

4.3. Our algorithm for random model generation

When generating a model, answers to the following questions define a unique stochastic game:

- How many states does the model have?
- Which states belong to which player?
- How many, and which actions does each state have?
- How many transitions does an action have, where do they lead, and how probable is the transition?
- What is the initial state?
- What is the set of targets T?

We use Algorithm 1 to create any random stochastic game that is connected from the initial state. After initialization, the algorithm has two phases: the forward and the backward procedure. During initialization, we generate a random number n and create a set of states $\mathsf{S} := [n]$. Also, we distribute the states at random between Maximizer and Minimizer. In the forward procedure, we iterate over every state $\mathsf{s} \in \mathsf{S}$ and make sure that a previous state s' is connected to it by providing an action with positive transition probability to s from s'. This guarantees that the initial state can reach every state in the stochastic game. The backward procedure then adds arbitrary actions to arbitrary states to enable generating every possible SG $\in \mathcal{G}_{Reachable}$.

To generate the actions of a state, we use Algorithm 2. It receives a state-action pair (s,a) where $\sum_{\mathsf{s}' \in \mathsf{S}} \delta(\mathsf{s},\mathsf{a},\mathsf{s}') < 1$. It then increases the transition probability of a randomly selected

state $\mathsf{s}' \in \mathsf{S}$ where $\delta(\mathsf{s},\mathsf{a},\mathsf{s}') = 0$. This is repeated until $\sum_{\mathsf{s}' \in \mathsf{S}} \delta(\mathsf{s},\mathsf{a},\mathsf{s}') >= 1$ or there is no state s' that is not yet reached by (s,a). In case $\sum_{\mathsf{s}' \in \mathsf{S}} \delta(\mathsf{s},\mathsf{a},\mathsf{s}') < 1$ holds but the state-action pair has a positive transition probability to every state in S, we increase the most recently increased transition probability such that the resulting distribution is a probability distribution. If we reach $\sum_{\mathsf{s}' \in \mathsf{S}} \delta(\mathsf{s},\mathsf{a},\mathsf{s}') > 1$, we reduce the transition probability we increased most recently. After applying Algorithm 2, (s,a) is a valid transition distribution whose probabilities sum up to 1.

Algorithm 1 Generating random models connected from initial state

Output: Stochastic game SG where the initial state is connected to any $\mathsf{s} \in \mathsf{S}$

1: Create S with a random $n \in \mathbb{N}$
2: Partition S uniformly at random into $\mathsf{S}_\Box$ and $\mathsf{S}_\bigcirc$
3: Enumerate $\mathsf{s} \in \mathsf{S}$ in any random order from 0 to n-1
4: Set s_0 to the state with index 0
5: Set $\mathrm{T} = \{s_{n-1}\}$
6: **for** $\mathsf{s} = 1 \to n-1$ **do** * Forward Procedure * \
7: **if** s does have an incoming transition **then** Continue (Skip iteration)
8: **else**
9: Pick any state s' with index smaller than s
10: Create an action a that starts at s'
11: Assign to (s, a) a positive probability of reaching s
12: Create a valid probability distribution for (s, a) by applying FillAction(s', a)
13: Add a to $\mathsf{Av}(\mathsf{s}')$
14: Add a to A
15: **for** $\mathsf{s} = n-1 \to 0$ **do** * Backward Procedure * \
16: Pick a random number $m \in \mathbb{N}$ * Decide at random how many actions to introduce * \
17: **if** $|\mathsf{Av}(\mathsf{s})| = 0$ **then** $m \leftarrow \max\{m, 1\}$ * Every state must have at least one action * \
18: **for** $i = 1 \to m$ **do**
19: $(\mathsf{s}, \mathsf{a}_i)$ = FillAction(s, a_i)
20: Add a_i to $\mathsf{Av}(\mathsf{s})$
21: Add a to A

Now that we have introduced our algorithm for randomly generating models, we need to prove two things: We need proof of the algorithm's correctness to ensure that the algorithm only generates formally correct stochastic models, and we need to prove that we can generate any model in *connectedSG*. We start with the former:

Lemma 4.3.1. *Algorithm 1 creates formally correct stochastic games.*

Proof. S is finite since its size is determined by a random number $n \in \mathbb{N}$ (line 1). Line 2 ensures that we have a partition of S into $\mathsf{S}_\Box$ and $\mathsf{S}_\bigcirc$. Next, line 3 and 4 provide an initial state and a

Algorithm 2 FillAction(s, a)

Input: outgoing state s, action a
Output: action a that has a valid underlying transition probability distribution

1: **repeat**
2: Pick a random state s' where $\delta(s,a,s') = 0$
3: Increase $\delta(s,a,s')$ by a random number $\in (0,1]$
4: **until** either $\sum_{s \in S} \delta(s,a,s') \geq 1$ or $\forall s \in S : \delta(s,a,s') > 0$
5: **if** $\sum_{s \in S} \delta(s,a,s') > 1$ **then**
6: Decrease the most recently modified $\delta(s,a,s')$ so $\sum_{s \in S} \delta(s,a,s') = 1$
7: **else if** $\sum_{s \in S} \delta(s,a,s') < 1$ **then** * Loop terminated because $\forall s \in S : \delta(s,a,s') > 0$ * \
8: Increase the most recently modified $\delta(s,a,s')$ so $\sum_{s \in S} \delta(s,a,s') = 1$
return (s, a)

target. Thus, we only need to argue that Av is truly a mapping of $S \to 2^A$ and that the transition function yields a probability distribution. When we introduce state-action pairs (line 9 and line 18), we introduce a new action that we add to A. Also, we add the state-action pair to Av (line 12 and 19). Thus, any Av is function of $S \to 2^A$.

Next we need to prove that for every $s \in S$ and every action $a \in Av(s)$ the transition function $\delta(s,a)$ yields a probability distribution. In other words we need to validate that (i) for every state $s' \in \mathit{states} : \delta(s,a,s') \in [0,1]$ and (ii) $\sum_{s' \in S} \delta(s,a,s') = 1$ are true.

To prove (i), note that whenever we introduce an action a to the set of enabled actions $Av(s)$ of a state $s \in S$, we have $\delta(s,a,s') = 0$ for all $s' \in S$. We increase $\delta(s,a,s')$ in line 10 of Algorithm 1 and lines 3 and 8 of Algorithm 2. We increase transition probabilities by numbers in $[0,1]$. Due to the condition in line 2 of Algorithm 2, $\delta(s,a,s')$ can only be increased a second time in line 8 of Algorithm 2. However, per state-action pair (s,a) with $a \in Av(s)$ line 8 of Algortihm 2 may be executed only once and we increase only by a $\Delta > 0$ such that $\delta(s,a,s') + \Delta + \sum_{s'' \in S \setminus \{s'\}} \delta(s,a,s'') = 1$. Since every state $s'' \in S \setminus \{s'\}$ was increased only once, $\delta(s,a,s') + \Delta \leq 1$.

To prove (ii), note that every pair (s,a) where $a \in Av(s)$ is applied to Algorithm 2. Once the loop (line 1-4) terminates, it either holds that (a) for every state $s' \in S : \delta(s,a,s') > 0$ or that (b) $\sum_{s' \in S} \delta(s,a,s') \geq 1$. If $\sum_{s' \in S} \delta(s,a,s') = 1$ the algorithm terminates. Thus, assume that this case does not hold. In case (a) line 8 increases the most recently added triple (s,a,s') by a $\Delta > 0$ such that $\delta(s,a,s') + \Delta + \sum_{s'' \in S \setminus \{s'\}} \delta(s,a,s'') = 1$. In case (b) $\sum_{s' \in S} \delta(s,a,s') > 1$. Due to the exit condition of the loop (line 4), without the most recently increased transition (s,a,s') the sum of the probabilities must be below 1, i.e. $\sum_{s'' \in S \setminus \{s'\}} \delta(s,a,s'') < 1$. Thus, there is a $\Delta \in (0,1), \Delta < \delta(s,a,s'')$ such that $\delta(s,a,s') - \Delta + \sum_{s'' \in S \setminus \{s'\}} \delta(s,a,s'') = 1$. We decrease (s,a,s') by this Δ. In conclusion, every pair (s,a) that is provided to Algorithm 2 yields a valid transition distribution where $\sum_{s' \in S} \delta(s,a,s') = 1$. Thus, Algorithm 1 generates formally correct

stochastic games. □

To show that this Algorithm can truly create any SG $\in \mathcal{G}_{Reachable}$, we consider the following lemma:

Lemma 4.3.2. *Let $\mathcal{G}_{Algo}$ be the set of stochastic games that Algorithm 1 can produce. Then $\mathcal{G}_{Algo} = \mathcal{G}_{Reachable}$.*

Proof. **Show** $\mathcal{G}_{Algo} \subseteq \mathcal{G}_{Reachable}$:

For this statement to hold, any SG $\in \mathcal{G}_{Algo}$ must be connected from the initial state. Proof by induction over the indices i of the states along their enumeration assigned during Algorithm 1:

Basis : $i = 0$: s_0 is the initial state. The initial state reaches itself within 0 steps.

Hypothesis: Let i be arbitrary but fixed with $i \leq n-1$, where $n = |\mathsf{S}|$. For every $j \leq i$ it holds that s_0 can reach s_j.

Step : $i \leftarrow i+1$: Due to the forward procedure it holds that

$$\exists \mathsf{s}_j \in \mathsf{S}, j < i, \mathsf{a} \in \mathsf{Av}(\mathsf{s}_j) : \delta(\mathsf{s}_j, \mathsf{a}, \mathsf{s}_i) > 0$$

However, according to our hypothesis s_0 is connected to s_j and thus also to s_i.

Show $\mathcal{G}_{Reachable} \subseteq \mathcal{G}_{Algo}$:

Pick an arbitrary but fixed stochastic game SG $\in \mathcal{G}_{Reachable}$. Next, we show that there is a run of our algorithm that will return a stochastic game SG$' \in \mathcal{G}_{Algo}$ where SG$'$ is an isomorphism to SG. Thus, G' and G are the same except for the state enumeration and the labels of the actions.

For this, we need several statements to hold at once:

1. The number of states in SG and SG$'$ is equal.
2. The partition of S to $\mathsf{S}_\Box$ and $\mathsf{S}_\bigcirc$ is the same for SG and SG$'$.
3. SG and SG$'$ have the same initial states and targets.
4. All state-action pairs in SG and SG$'$ yield equal probability distributions in δ.
5. Every state in SG and SG$'$ has the same actions.

We decide randomly on the number of states in line 1 of Algorithm 1 and partition S into $\mathsf{S}_\Box$ and $\mathsf{S}_\bigcirc$ at random in line 2 of Algorithm 1. Thus, for every SG $\in \mathcal{G}_{Reachable}$ there exists SG$' \in \mathcal{G}_{Algo}$ that have the same number of states to which there is an enumeration such that they are partitioned equally in both stochastic games. Since the states can be arranged in any order, we pick the initial state of G' such that is the same as in G. All targets of SG can be mapped to the singular target of SG$'$ since they only have self-loops and behave identically to f of SG$'$. Thus, there exists a run where statements 1, 2, and 3 hold.

When using Algorithm 2 to create a probability distribution for a state-action pair (s,a), we increase transition probabilities until they sum up to 1. Thus, any summation $\sum_{\mathsf{s}'\in\mathsf{S}}\delta(\mathsf{s},\mathsf{a},\mathsf{s}')=1$ is possible. In consequence, an action may lead into arbitrary states, have an arbitrary number of positive transition probabilities between 1 and $|\mathsf{S}|$, and may have arbitrary probability distributions on the transitions as long as they sum up to 1. So out of all runs where statements 1, 2, and 3 hold, there also must be at least one run where statement 4 holds too.

Proving statement 5 is crucial to ensuring that every state may have an arbitrary positive number of actions. While the backward procedure enables every state to have many actions, we need to prove that games in $\mathcal{G}_{Reachable}$ where every state has few actions can also be generated by our algorithm. To prove the statement, note that each SG $\in \mathcal{G}_{Reachable}$ has a minimal set of state-action tuples such that the initial state may reach every state. Taking this set, we perform a breadth-first-search from the initial state to provide an enumeration of the states. If we iterate over the states following this enumeration, we can reproduce each of the actions in the minimal set during the forward process. Due to the enumeration, to each state s except for s_0, there is a state with a smaller index s' such that s' has an action a with a positive transition probability of reaching s. Since every other transition of (s',a) can lead into arbitrary states and the probability distribution of (s',a) can be arbitrary, we can recreate the minimal set of state-action tuples in SG$'$.

The remaining state-action pairs of SG can be added to SG$'$ during the backward process, where every state may add arbitrarily many actions with arbitrary transition distributions. □

Note that although $\mathcal{G}_{Algo}=\mathcal{G}_{Reachable}$, in general, Algorithm 1 does not sample $\mathcal{G}_{Reachable}$ uniformly at random. Due to the forward procedure, states with smaller indices tend to have more actions than states with higher indices. Additionally, creating the transition distributions as described in Algorithm 2 favors state-action pairs to have few transitions. If we pick the transition probabilities between $(0,1]$ uniformly at random, around $83,33\%$ of all actions have two or three transitions with positive probability and none with one transition.

4.4. An alternative to handcrafted models

Since the purpose of randomly generated models is to sample from a space as big as possible, we have few insights into the structure of the model. Thus, this method is not very well suited if one wants to analyze the behavior of an algorithm on very specialized models. Traditionally, this is where one would build handcrafted models. However, handcrafting parameterizable prism files allows only for very restricted structural changes in the models. Therefore, we have added the option to create models via code in PRISM. On one hand, this allows us to expose parameters to the user that influence more structural properties than it is possible with handcrafted models. On the other hand, we can prepend models to other models. This means that we unify a given model SG with another game SG$'$ we create at runtime to one new stochastic game SG$''$. The initial state

of SG′ is the initial state of SG″, and the targets of SG are the targets of SG″. All state-action pairs from both games are used in SG″, but whenever SG′ would reach a target, it reaches instead the initial state of SG. Since SG never reaches SG′, the runtime of any topological variant of an algorithm is the sum of the runtime to solve both models. Prepending models allows us to analyze how much an added component influences algorithm performance.

5. Implementing randomly generated stochastic games

In this chapter, we discuss how our implementation of random generation differs from the theory described in Chapter 4 as well as how to use and extend our implementation.

5.1. Parameters and guidelines for model construction

We have implemented a constrained version of Algorithm 1 from Section 4.3 to randomly generate models. The real-world implementation has to be constrained due to the natural restriction that a computer cannot generate arbitrarily big stochastic games and arbitrarily small transitions due to finite memory. Additional constraints on the real-world implementation are the pseudo-randomness while taking decisions as well as floating-point machine precision.

However, usually, the users also want some control over the properties of the resulting models like the number of states. If all model properties are differing too much, it is very hard to deduct why an algorithm performs differently on two models. For example, it is very likely that if we would always randomize the number of states of a model, the models would differ so much in their number of unknown states that all other structural differences would have a comparatively diminishing impact on algorithm performance.

Thus, we provide several parameters which we do not randomize. Instead, we give the user the option of providing a value for the parameter or using our default parameters. The parameters in Subsection 5.2 provide an overview of what we do not randomize in our implementation. Of course, it is possible to wrap a script around our generation implementation that randomizes the parameters to increase the variation of generated models.

Limits and additional guideline options

Although the parameters we expose for random generation constrain some structural properties of the resulting models, there are many structural properties we can only barely influence. For example, there is no obvious way how to influence the size of SCCs that a model has or to guarantee that every state in a model has a certain number of actions when using Algorithm 1.

For this, we provide the option in our implementation to use other scripts for the model generation that are wrapped into PRISM files. We refer to these scripts as guidelines. We

provide two additional guidelines to our random-generation procedure described in Section 4.3: A procedure that controls how many actions each state has and another one that provides a guarantee on the size of SCCs in the model. Both guidelines generate model SG $\in \mathcal{G}_{Reachable}$ but are not able to create any game in $\mathcal{G}_{Reachable}$.

The guideline that controls how many actions a state has is called *RandomTree* because it creates a treelike graph structure where the initial state is the root. Every node of the tree has k actions and at most k children, where k is a parameter. Every action has an assigned child to which it has a positive transition probability. The rest of the probability distribution of the action is assigned at random. An inner node of the tree may have less than k children if adding k children would exceed the requested number of states n for the model. Also, leaves are not required to have k actions. Their actions are only introduced during the backward process and enable the generation of end components.

We refer to the guideline that controls the SCCs of a model by *RandomSCC*. The procedure requires a minimal and maximal size boundary $[a,b]$ for every SCC and the total number of states in the stochastic game n. First, we create subgames of a randomly chosen size in $[a,b]$. The subgames are created by the algorithm described in Section 4.3. We then use Tarjan's algorithm for strongly connected components [Tar72] to identify the SCCs of the created subgame. Next, we unify all the SCCs of the subgame by using the topological enumeration Tarjan's algorithm provides. We circularly connect the SCCs following the enumeration, making the whole subgame an SCC. Next, we make sure that the subgame is connected to the rest of the stochastic game by making sure a previously created subgame has an action leading into this subgame. We repeat this procedure until we have at least n states in the stochastic game, resulting in a stochastic game SG $\in \mathcal{G}_{Reachable}$ where the user has an easy way of controlling the size of the SCC in SG.

5.2. A manual on how to use our implementation

In this section, we describe the implementation details and provide guidance for using, understanding, and extending our implementation. The random generation is split into multiple Python modules to maximize extendability and readability. We recall that a Python module is a file with a script. All relevant Python modules are located in the folder "random-generated-models" in the GitHub Repository `https://github.com/ga67vib/Algorithms-For-Stochastic-Games`. We provide a description of how to use the modules and how to extend them in case the reader wants to.

Generating models

If you want to generate a model, you can run the `modelGenerator.py`-script. This script can receive several parameters to guide the generation process and constrain which models can be created.

You can use `python modelGenerator -help` to see all possible parameters and what they do. Here is the exhaustive list of parameters:

- outputDir: Where to put the resulting model?
- size: How many states should the generated model have??
- numModels: How many models to create?
- minIncomingActions: The minimum of how many actions should lead into a state during the forward procedure of algorithm 1.
- maxIncomingActions: The maximum of how many actions should lead into a state during the forward procedure of algorithm 1.
- maxBackwardsActions: The maximum of how many actions should be added to a state during the backward procedure of algorithm 1.
- guideline: Which guideline to use when generating the model. Will use by default Algorithm 1. Alternatives are the "RandomTree" and "RandomSCC" guidelines that are described in Section 5.1.
- smallestProb: What is the smallest probability that is allowed to occur?
- backwardsProb: If a random probability is generated below this threshold, the state receives no actions in the backward procedure.
- branchingProb: If a random probability is generated below this threshold, the state-action pair leads into only one state with probability 1.
- maxBranchNum: How many positive transition probabilities may one action have at most?
- forceUnknown: A switch. If set to true, a new target and sink are introduced that effectively multiply the value of each state by 0.9. This ensures that no additional targets are created during pre-computation. We introduce the switch to ensure that the generated model is not solved mostly during pre-computation.
- maximizerProb: What is the probability that a state belongs to the Maximizer?
- minSCCSize: What is the smallest size an SCC may have (only for RandomSCC guideline)?
- maxSCCSize: What is the largest size an SCC may have (only for RandomSCC guideline)?
- fastTransitions: When used, keep a stack of shuffled state indices that is popped whenever a successor for a transition is needed. Transition goals are more predictable but generated significantly faster.

The generated model will be named by the parameter values and will be stored in the specified directory or `generatedModels` by default.

To generate multiple models with different parameter settings, we have used the simple script `massModelGenerator.py` which calls `modelGenerator` various times with the different sets of parameters.

Module explanation

Next, we explain the functionality of every module. The module `modelGenerator.py` is responsible for calling the module that generates the stochastic game and translating it into a .prism file. Furthermore, it is the interface for users.

The creation of the stochastic game is outsourced to another python module. The random generation as described in Algorithm 1 happens in module `graphGenerator.py`. This is also the base class for other procedure guidelines. Both the RandomTree guideline - implemented in model `treeGraphGenerator.py` - and the RandomSCC guideline - implemented in module `sccGraphGenerator.py` - inherit from `graphGenerator`.

The module `graphGenParams.py` contains a class that contains all the parameters any graph generator requires.

Lastly, there is a module called `randomStateGetter.py`. This class implements various randomPickers. A randomPicker selects a state of the state-space randomly. For example, this is used to decide which state should have an action to the currently introduced state. Different implementations allow us to guide the selection process to achieve different distributions of actions and transitions. Each generator receives two randomPickers as part of their input parameters: One that decides which state should receive an outgoing action, and another one that decides where a transition should lead. By default, the randomPicker picks any state $\in [\mathsf{s}_0, \mathsf{s}_m]$ randomly at uniform, where m is a parameter.

Module extension

If you are interested in implementing a new way of generating random models by providing new guidelines, all you have to do is create a new module that inherits from `GraphGenerator` and implement the functionality. The `generateGraph()` call must receive a `GraphGenParams` object and return nothing. After `generateGraph()`, the `GraphGenerator` instance should hold all the information about the game in its class variables.

6. Tools for algorithm analysis

To facilitate the algorithm performance analysis, we track statistics about the models we use and the algorithms that solve them. We refer to every category of data that we track as *features*. The algorithm features we track are the time it requires to solve the problem, the iterations needed in case we use a value-iteration-based algorithm, and the value it has computed for correctness checks. The model features we track along with their names we use in the experimental section are:

Feature Name	Description	Rel.
State – related		
NumStates	Number of states	
NumTargets	Number of targets	×
NumSinks	Number of sinks	×
NumUnknown	Number of unknown states	×
NumMaxStates	Number of Maximizer states	×
NumMinStates	Number of Minimizer states	×
Actions – related		
NumActions	Total number of actions	
NumMaxActions	Maximal occurring number of actions per state	
NumProbActions	Number of non-deterministic actions	×
AvgNumActionsPerState	Average number of actions per state	
Transitions – related		
NumMaxTransitions	Number of maximum transitions per action	
SmallestTransProb	Smallest occurring positive transition probability in the model	
AvgNumTransPerAction	Average number of transitions per action	

Feature Name	Description	Rel.
MEC – related		
NumMECs	Number of MECs	
Biggest-/ SmallestMEC	Size of biggest / smallest MEC	×
AvgMEC, MedianMEC	Average MEC size and Median MEC size	×
SCC – related		
NumSCCs	Number of SCCs	
Biggest-/ SmallestSCC	Size of biggest / smallest SCC	×
AvgSCC, MedianSCC	Average SCC size and median SCC size	×
Biggest-/ SmallestSCC	Size of biggest / smallest SCC	×
MaxSCCDepth	Longest chain of SCCs from the SCC of the initial state	
NumSCCNonSingleton	Number of SCCs with size at least 2	
SmallestSCCNonSingleton	Smallest SCC with size greater than 1	
AvgSCCNonSingleton	Average SCC size with cardinality greater than 1	×
Path – related		
NearestTarget	Shortest Path from initial state to the nearest target	
FurthestTarget	Shortest Path from initial state to the furthest target	

Table 6.1.: A summary of all features we track in every model. If a × symbol is present in the last column of the feature, the feature is measured absolutely, but we display it relative to a fitting measure.

Features with a × symbol are measured absolutely but displayed relatively. When referring to the relative value, we use the % sign instead of the "Num"-prefix (e.g. Unknown% instead of NumUnknown). Every relative feature but NumProbActions is displayed relative to $|S|$ (NumStates). NumProbActions is displayed relative to NumActions. We found it easier to extract knowledge from relatively displayed features due to their independence of the model size.

Visualization

Tracking all these features requires tools to visualize and summarize the collected data since otherwise, the raw data is overwhelming. We have implemented a bare-bones toolset to facilitate the analysis.

Data visualization script - data loading

First, we load the tracked data and select the features we are interested in. We allow here to include filters to remove uninteresting data based on feature values. We also check for errors and wrong values in the data loading phase. To assert whether a value is correct, we need a reference value whose result we believe to be true. We usually use $\mathbf{SI_{LP}}^{\mathbf{T}}$ or **WP** as references since they tend to have the least timeouts and thus provide a good reference.

Data visualization script - handling missing data

Next, we need to handle data that does not contain numerical values - for example, the algorithms that got timeouts on certain models.

One way to handle this situation is by only analyzing models where every algorithm finishes computation. However, then we would risk losing information about models where one algorithm would perform way better than another algorithm.

Instead, we assign fixed values for missing features. For model features, we usually use 0 as a replacement value. If an algorithm fails to provide the correct value in time, we set its time and iterations count to a penalty value. These penalty values should be higher than any truly occurring value to easily calculate and visualize which algorithms performed best. For time, we usually use the time limit as penalty, and for iterations, we use 10 million iterations as penalty.

The problem with introducing these penalty values is that they influence the data distribution. For visualization, the graphs can end up skewed because most non-penalty data points are significantly smaller than the penalty, and so it becomes harder to observe trends in graphs. In these cases, we zoom in on the majority of the non-penalty data to be able to observe trends but take note of the outlying values. Also, many visualization methods like heatmaps or statistical tests perform operations on the data. Thus, incorrect values can falsify the correlation between features. Thus, when working with heatmaps, we only include those models that are solved by every algorithm.

7. Results

In this chapter, we present first the technical details of our experimental setup. Next, we analyze the structural biases of the real case studies and compare them to the structural biases of randomly generated models we have created. Furthermore, we evaluate our algorithm for random generation in terms of scalability and runtime performance. Then, we compare algorithm performance on the real case studies as well as our randomly generated models and investigate which model features influence the performance of the algorithms.

7.1. Experimental setup

First, we discuss the details of our experimental setup. Various algorithms we consider were already implemented in PRISM-games [Kwi+20]. We extended PRISM-games by the algorithms $\mathbf{SI_{LP}}$, $\mathbf{SI_{SI}}$, and $\mathbf{BVI^{PT}}$. Moreover, for $\mathbf{BVI^{PT}}$ and $\mathbf{SI_{SI}}$, we added linear equation solving with Gurobi version 9[1], which we used with an academic license. For $\mathbf{SI_{LP}}$, we use Gurobi version 9 too. Our code is available in the GitHub repository `https://github.com/ga67vib/Algorithms-For-Stochastic-Games`.

Technical details

We conducted the experiments on a server with 64 GB of RAM and a 3.60GHz Intel CPU running Manjaro Linux. We always use a precision of $\varepsilon = 10^{-6}$. The timeout was set to 15 minutes, and the memory limit was 6 GB for all models except for the set of large models (see Subsection 7.3.1), where the timeout was set to 30 minutes and the memory limit to 36 GB.

7.1.1. Case studies

We consider case studies from four different sources:

(i) real case studies, of which all but "teamform" ([Che+11a]) were already used in [Kře+20], and which are mainly from the PRISM benchmark suite [KNP12]. For a detailed description of the real case studies, see [Kře+20, Appendix C.1]. We omit models that are already solved by pre-computations. With variations of the model parameters, we obtain 18 models.

[1] https://www.gurobi.com/

(ii) several handcrafted corner case models: Haddad-Monmege (an adversarial model for value iteration from [HM18]), BigMec (a single big MEC), and MulMec (a long chain of many small MECs), the latter two both being from [Kře+20].

(iii) 300 randomly generated models generated by Algorithm 1 and our additional guidelines from Subsection 5.1. For the exact parameters we have used to generate our models See Appendix A. Without models solved by pre-computations, we obtain 284 models.

(iv) large models (models with at least 100,000 states) with a structure similar to the RandomTree guideline (see Section 5.1) and generated at runtime with our alternative to handcrafting as described in Section 4.4. The difference to models generated with the RandomTree guideline is that every inner tree node has three deterministic actions, and only the leaves have probabilistic actions that lead into either the next SCC, the target, the sink, or the root of the tree. The models have at least 100,000 states and at most 5 million states. For each size, there is one model where the whole state space is strongly connected, and another one where there the state space is divided into five equally large SCCs.

7.1.2. Plot overview

We provide a short description of each type of plot we use in this chapter:

Box plots

A box plot provides an overview of the spread and skewness of the model features of our model sets. See Table 6.1 for an overview of all model features we track and an explanation of the abbreviations. The orange line marks the median of a feature in all models, and the green triangle marks the average. The bounds of the boxes mark the 25 and 75 percentile, and the extended lines mark the last data point included in the interquartile range times the factor 1.5. Dots outside the whiskers represent outliers that differ significantly from the rest of the dataset. The plots are grouped by and colored into the following categories:

- Green outlines are for properties related to states.
- Blue outlines are for properties related to actions.
- Cyan outlines are for properties related to transitions.
- Red outlines are for properties related to MECs.
- Orange outlines are for properties related to SECs.

Line plots for accumulated algorithm performance

To provide a general overview performance of all stochastic game algorithms, we use line plots. The plot depicts the number of solved benchmarks (x-axis) and the time it took to solve them (y-axis). For each algorithm, the benchmarks are sorted ascending by verification time. A line stops when no further benchmarks could be solved. Intuitively, the further to the bottom right a plot is, the better; where going right (solving benchmarks) is more relevant. The legend on the right is sorted by the performance of the algorithms in descending order. Note that this plot has to be interpreted with care, as it greatly depends on the selection of benchmarks.

Scatter plots for algorithm performance

While line plots compare models solved and accumulated performance, scatter plots allow us to compare algorithm performance model by model. Each point in the graph is a model. The x-axis marks the time/iterations one algorithm requires to solve a model, and the y-axis marks the respective time/iterations of the compared algorithms. If a point is below the diagonal, the algorithm on the x-axis required more time to solve it than the corresponding algorithm on the y-axis and vice versa. The two lines next to the diagonal mark the case where one algorithm was twice as fast as the other.

1-dimensional scatter plots

We use 1-dimensional scatter plots to visualize whether structural features favor one event over another. In some cases, we want to analyze how two sets of events $\mathbf{A}, \mathbf{B}$ correlate to model features. Usually, we use **B** as the complement to **A**. An example for such a pair is: **A** contains all the models where algorithm X was 1.5 times faster than algorithm Y. **B** contains all the models where algorithm X was not 1.5 times faster than algorithm Y. We study the properties of the models in **A** and the properties of the models in **B** by scattering the feature values of models in **A** against scattering feature values of models in **B**. When using this plot, we try to identify whether models in **A** distribute differently along the spectrum of feature values than **B**.

7.2. Model analysis results

In this section, we use our analysis tools to investigate structural biases in the real case studies and in our randomly generated models. First, we want to learn about the feature distribution of the real case studies we have. For this, we use a box plot for each feature in Figure 7.1. The box plots provide the following insights:

- On average models have 2 actions per state and 1.5 transitions per action (see AvgNumActionsPerState and AvgNumTransPerAction).

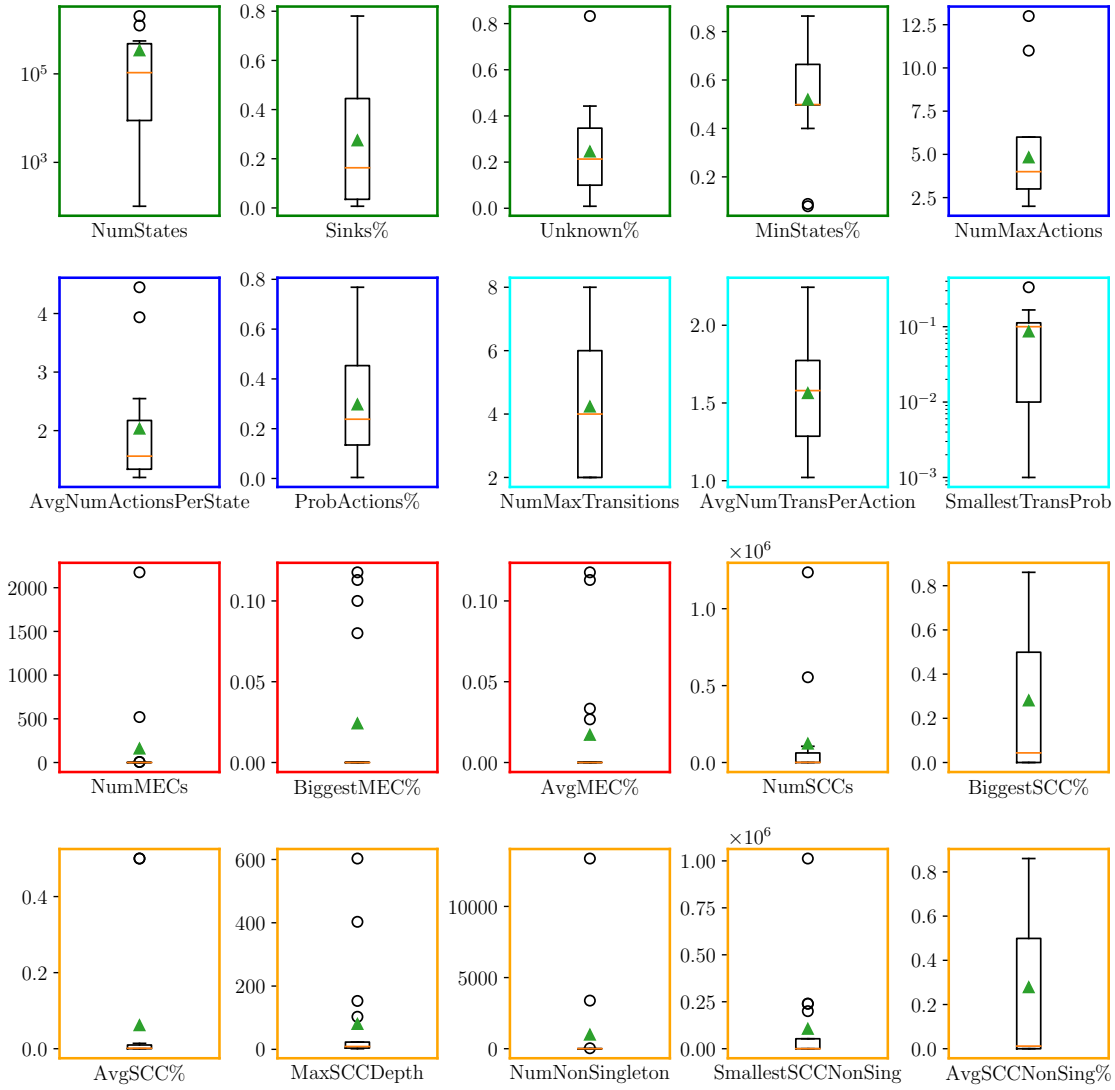


Figure 7.1.: A Box plot of the feature distribution of the real case studies. The description of how to read the plot is provided in Section 7.1.2.

- The plot for NumUnknown shows that usually at least 50%, and on average 80% of the states of the models are trivial, and their value is computed during pre-computation with simple graph algorithms.
- Generally, the Maximizer controls as many states as the Minimizer.

- The plot for NumProbActions indicates that usually, around 70 to 85% of all actions are deterministic.
- Most models do not contain end components as NumMECs shows.

By furthermore inspecting the maximal and minimal occurring values of each feature we obtain that the smallest occurring transition probability is 0.001.

We also use the information to draw conclusions about which structural cases do not appear in the real case studies. None of the models contain these cases:

- Models with numerous actions per state,
- models with numerous transitions per action, or
- models with very small transition probabilities.

However, this does not say that these properties are necessarily unrealistic. Instead, they did not appear in problems people have modeled so far. After analyzing the real case studies, we now investigate the biases of the randomly generated models we have generated with Algorithm 1. Figure 7.2 contains the box plots for our randomly generated models.

The biases we read from this plot are:

- Unknown% indicates that on average, 38.5% of the state-values are obtained during pre-computation. Sinks% shows that almost all known states are sinks.
- With the chosen parameters, our algorithm generates models with 2 actions per state on average and 2 transitions per action on average (see AvgNumActionsPerState and AvgNumTransPerAction). However, our parameters allow us to change the number of actions and transitions per state.
- NumNonSingleton shows that in almost all cases, there is only one strongly connected component. This is because we uniformly randomize where a transition may lead. Thus, it is likely that big SCCs are formed. If necessary, the RandomSCC guideline can control the size of the SCCs.
- NumMECs indicates that there is usually either one MEC or none at all. Also, the MECs tend to have very few states (usually no more than 2). However, there are parameter configurations that favor generation of larger MECs. For example, setting the parameters in such a way that the actions are deterministic and adding one or multiple actions to every state in the backward procedure of Algorithm 1 creates models with a high tendency of forming few MECs that usually contain almost the whole state-space. Nevertheless, without providing a specific guideline, we have very limited control over the number and size of the MECs.

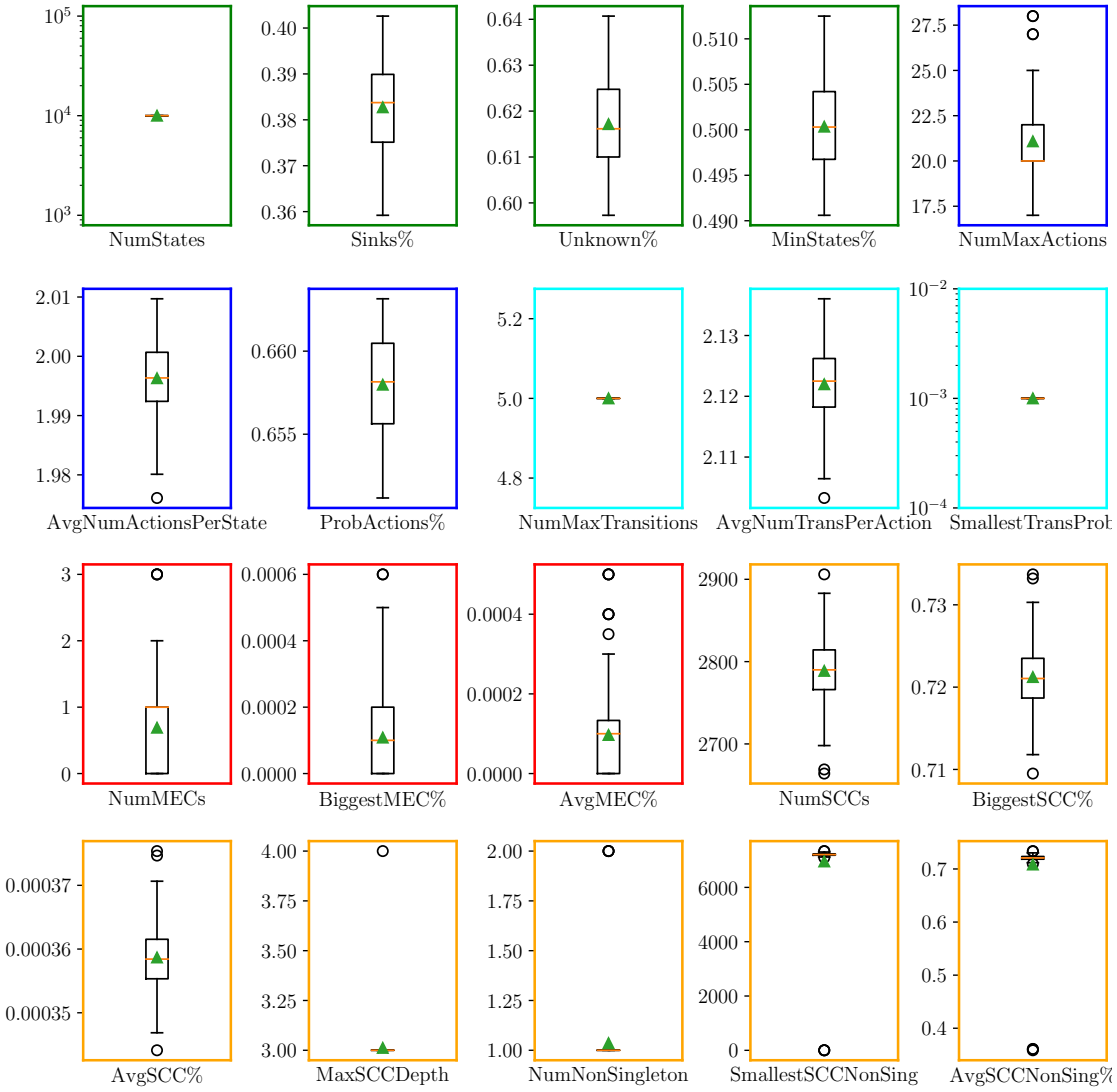


Figure 7.2.: A Box plot of the feature Distribution of models generated randomly with Algorithm 1. The description of how to read the plot is provided in Section 7.1.2. For each plot, 100 data points are used.

- Our random generation algorithm introduces a bias towards various properties like the number of SCCs or the average number of transitions per action. While on average models have only two actions, the maximal number of actions a state has is usually between 20 and 22 as NumMaxActions indicates. When analyzing the number of actions in relation to

the state index, states with many actions always have low indices.

When comparing the feature distributions of the real case studies and the distributions of the models generated by Algorithm 1, they have similar biases for many features. Both benchmarks tend to have few big SCCs, few actions per state, and few transitions per action. However, we are not restricted to creating models similar to the current case studies. The number of actions and transitions can be adjusted by parameters, and the how SCCs are formed can be influenced by guidelines, as described in Section 5.1.

To demonstrate this, we present in Figure 7.3 the feature distribution of a set of models we have created by using the RandomSCC guideline. Here, we constrain that every SCC should have 1000 to 2000 states. For the exact generation parameters, see Section A. Figure 7.2 contains the box plots for our randomly generated models.

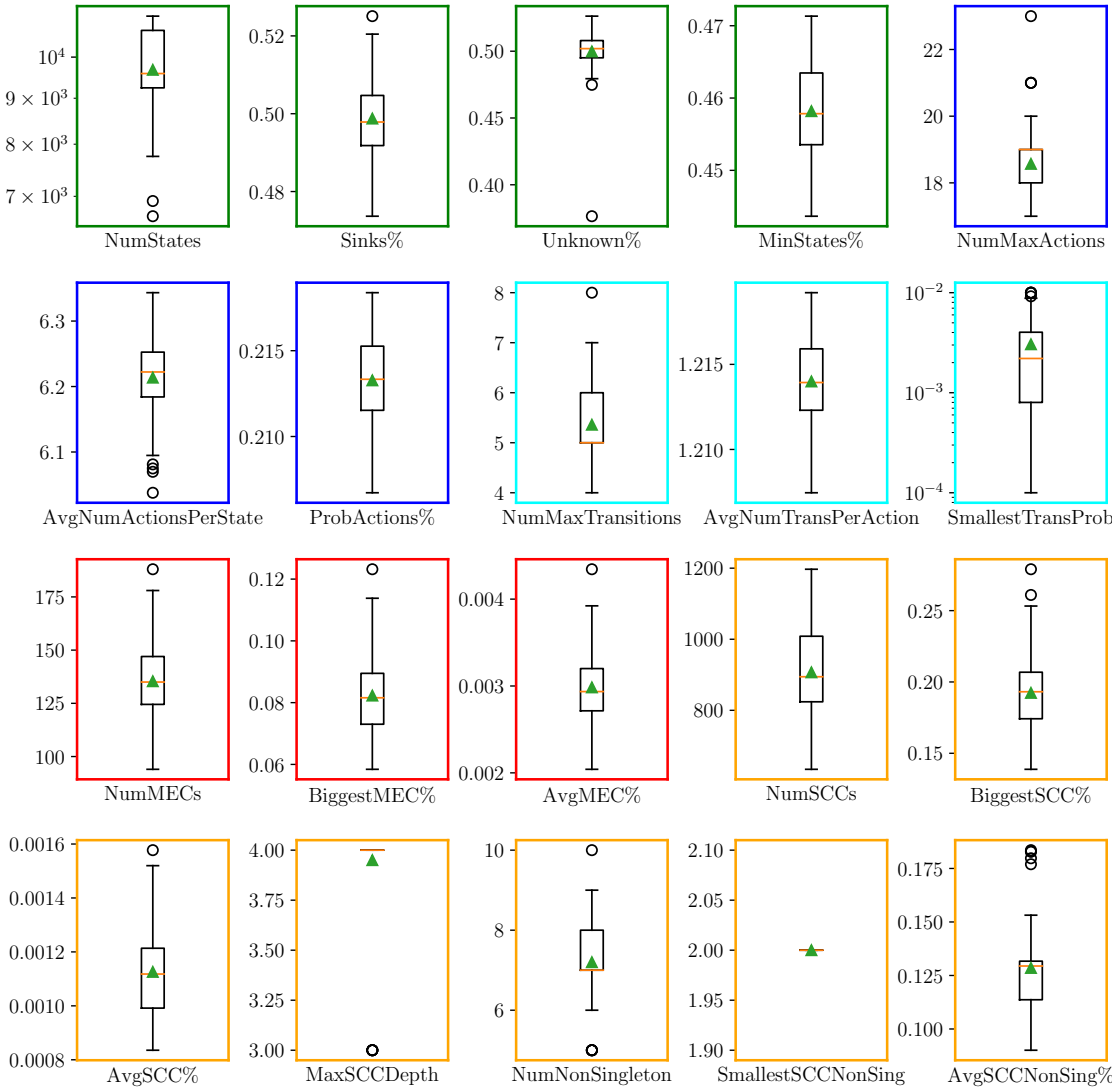


Figure 7.3.: Feature Distribution of randomly generated models by applying the RandomSCC guideline. The description of how to read the plot is provided in Section 7.1.2. For each plot, 100 data points are used.

Clearly, the guideline allows us to control the size of the SCCs. No SCC has more than 2000 states, which is the upper bound we provided. Also, MaxActions is not as biased as in Algorithm 1. However, this is only due to the SCCs being smaller. If every SCC had size 10000, the bias similar to random generation without guideline.

7.3. Evaluating our random model generation

Now that we have evaluated the models we generated, we evaluate the random generation algorithm itself in this section. We investigate the capabilities of generating large models and the required time to generate a model. Furthermore, we discuss difficulties we encountered in generating non-trivial models with our implementation as described in Chapters 4 and 5.

7.3.1. Generating large models

Note that all the randomly generated models have 10000 states. On one hand, we have chosen a fixed state size to make the results more comparable. On the other hand, PRISM was unable to parse large models in an adequate time. This is because every action of our randomly generated model is stored explicitly in the corresponding .prism file. PRISM requires a lot of time to parse these files. Parsing the randomly generated models with 10000 states with 2 actions per states takes up to 200 seconds and scales up non-linearly with the number of actions and states in our models. Since files with over one million states could not be parsed within an hour, we conclude that at the moment, PRISM is not able to handle our large randomly generated models.

Handcrafted models and real case studies achieve a larger state space since many .prism files are parameterized. Effectively, not every action is written out explicitly but is stored implicitly, allowing the model to be parsed faster. Generating models at runtime as we describe in Section 4.4 skips the state exploration process and creates large models fast. Thus, our benchmarking set for large models contains only handcrafted games we generate at runtime. While we also considered including real case studies with parameter configurations that create large models, we did not have the time to manually investigate the structural properties of all the parameterizable models. This investigation is crucial to identify whether certain performance tendencies may be due to structural biases of the benchmarking set. Therefore, all of our studies on large models are preliminary.

7.3.2. Time performance for random generation

Next, we evaluate the time our algorithm requires to generate a model.

Table 7.1 provides an overview of the time the algorithm needs to generate models with 10000, 100000, and 1000000 states and 1, 10, or 100 actions. Clearly, the generation process is becoming significantly slower for large models. This is mostly due to the way we generate actions (as described in Algorithm 2). When selecting a random state the action should lead into, we have to ensure this state is not yet included in Post of the current state-action pair. Thus, we subtract the states we already reach in the state-action pair from all states. Subtracting the sets for every transition poses the bottleneck in our generation process. To fix this issue, we provide the `-fastTransitions` switch. When used, we hold a stack of shuffled state indices that we pop whenever we need to generate a new transition. The switch makes the generation

Fast	(10e3, 1)	(10e3, 10)	(10e4, 1)	(10e4, 10)	(10e5, 1)	(10e5, 10)	(10e5, 100)
False	12s	44s	50min	3h 20min	>4h	>4h	>4h
True	2s	3s	7s	16s	45s	2min 30s	14min

Table 7.1.: Table of time that Algorithm 1 requires to generate one model for different numbers of states and actions per state. The "False" row shows the algorithm without the `-fastTransitions` switch, and the "True" row shows the time required if the switch is activated. The first number of the tuple in the header holds the number of states in the model, while the second is the `-minIncomingActions` parameter as described in Subsection 5.2.

slightly more predictable but decreases the time required considerably as 7.1 shows. Lastly, since PRISM requires more time to parse a model every time an algorithm must solve it than we need to generate the model, and since due to PRISM we are bound to using randomly generated models of size no larger than 50000 states, we conclude that the time performance is not an issue for now.

7.3.3. Increasing the number of unknown states

A reoccurring problem we encountered was that many models we generated randomly were solved entirely or mostly by simple graph algorithms. While in models generated by Algorithm 1 less than 40% of the states' values were solved during pre-computation with trivial graph algorithms, for the guidelines RandomSCC and RandomTree on average, more than 90% of the state space was computed trivially. If the entire or most of the state space is precomputed, the resulting problem is usually too easy to draw meaningful conclusions from it. To artificially make the problems generated with the RandomSCC or RandomTree guidelines harder, we set the forceUnknown switch to true. At the moment, this is equivalent to disabling the computation of states that can surely reach the target. The remaining known states are then either states unable to ever reach the target or states of the Minimizer that can choose to never reach the target.

7.4. Algorithm comparison results

In this section, we compare the algorithms introduced in Section 2.7 on real case studies, hand-crafted examples, and our randomly generated models to both evaluate the performance of the algorithms relative to each other and find correlations between model feature values and algorithm performance. After providing a broad overview of the algorithm performance, we compare the unoptimized value iteration extensions **BVI**, **OVI**, and **WP**. Next, we investigate whether the optimizations introduced in Subsection 2.7.4 improve their standard algorithm versions. Lastly, we compare the different approaches for strategy iteration.

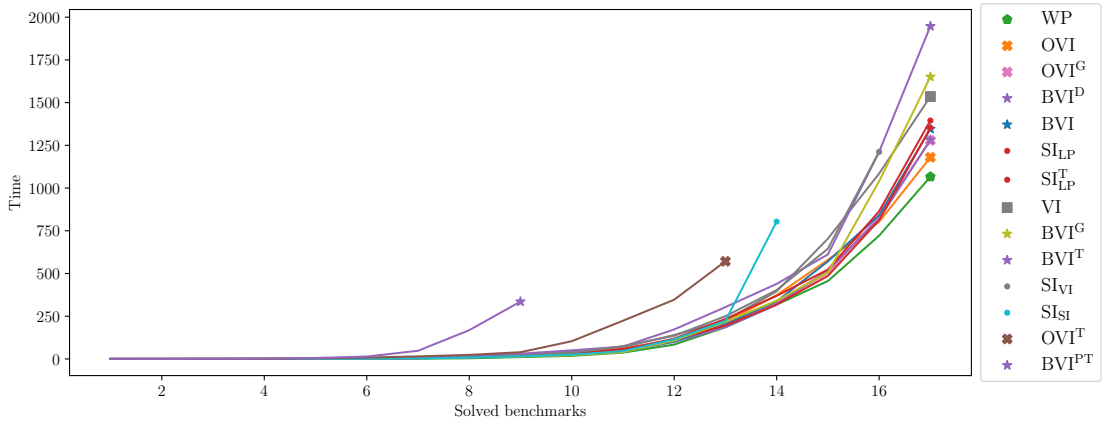
Since we refer to all the algorithms we analyze mostly by their abbreviations, we recall first the meaning of the abbreviations in Table 7.2.

Abb.	Full Name	Short description
VI	Value iteration	The standard value iteration with arbitrary precision.
BVI	Bounded value iteration	Value iteration with both lower and upper bound that provides a precision guarantee.
WP	Widest path bounded value iteration	Bounded value iteration with the widest path approach for solving end components.
OVI	Optimistic value iteration	Value iteration that guesses the upper bound to provide a precision guarantee.
$\mathbf{SI_{VI}}$	Strategy iteration with value iteration for MDPs	When the Maximizer strategy is fixed, the resulting MDP is solved with value iteration.
$\mathbf{SI_{LP}}$	Strategy iteration with linear programming for MDPs	When the Maximizer strategy is fixed, the resulting MDP is solved with linear programming.
$\mathbf{SI_{SI}}$	Strategy iteration with strategy iteration for MDPs	When the Maximizer strategy is fixed, the resulting MDP is solved with strategy iteration, yielding a DTMC which is solved with linear programming solvers.
T$*$	Topological optimization	Stochastic games are decomposed into their SCCs and solved following the topological enumeration.
G$*$	Gauss-Seidel optimization	Algorithms based on value iteration compute the value in-place.
D$*$	Deflation optimization	Deflating for value iteration variants with upper bounds is performed every 100 steps.
PT$*$	Precise & Topological optimization	A **BVI**-optimization. Like $\mathbf{BVI^T}$, but the values of each SCC are computed exactly.

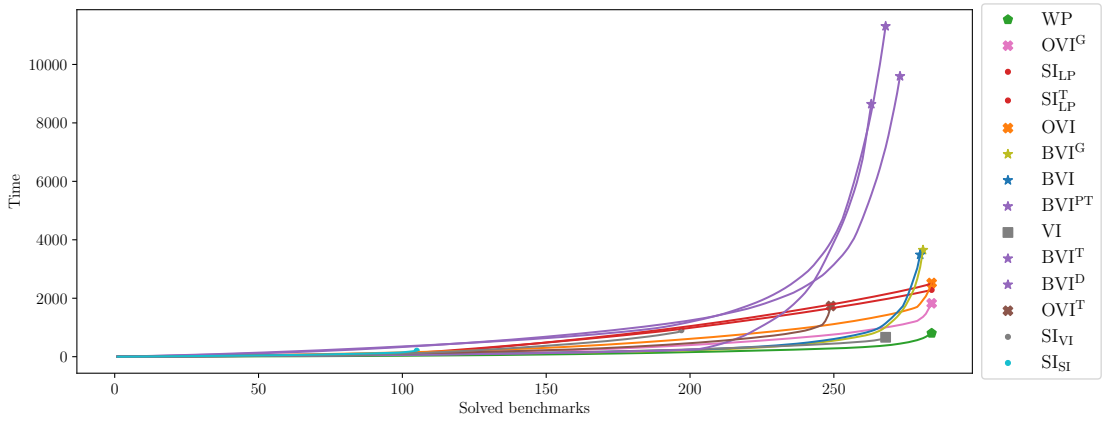
Table 7.2.: A summary which repeats the full names of the algorithm abbreviations we use paired with a short description. Optimizations are marked with a "*" symbol and are additions to other algorithms. For example, $\mathbf{BVI^T}$ is **BVI** with the topological optimization. For a detailed description of the algorithms please refer to Section 2.7.

To provide a general overview of the performance of all algorithms on our benchmarking sets, we present a line plot per benchmarking set in Figure 7.4.

We emphasize that since the times are displayed in accumulation and are very dependent on the models we use, the line plots only serve for rough overviews and provide insufficient information to compare algorithms that perform similarly.



(a) Accumulated algorithm performance overview on real case studies



(b) Accumulated algorithm performance overview on all randomly generated models.

Figure 7.4.: A line plot providing an overview of the accumulated algorithm performance. The plot depicts the number of solved benchmarks (x-axis) and the time it took to solve them (y-axis). For each algorithm, the benchmarks are sorted ascending by verification time. A line stops when no further benchmarks could be solved. Thus, the further to the bottom right a plot is, the better. Going right (solving benchmarks) is more relevant than having a low y-value. The legend on the right is sorted by the performance of the algorithms in descending order.

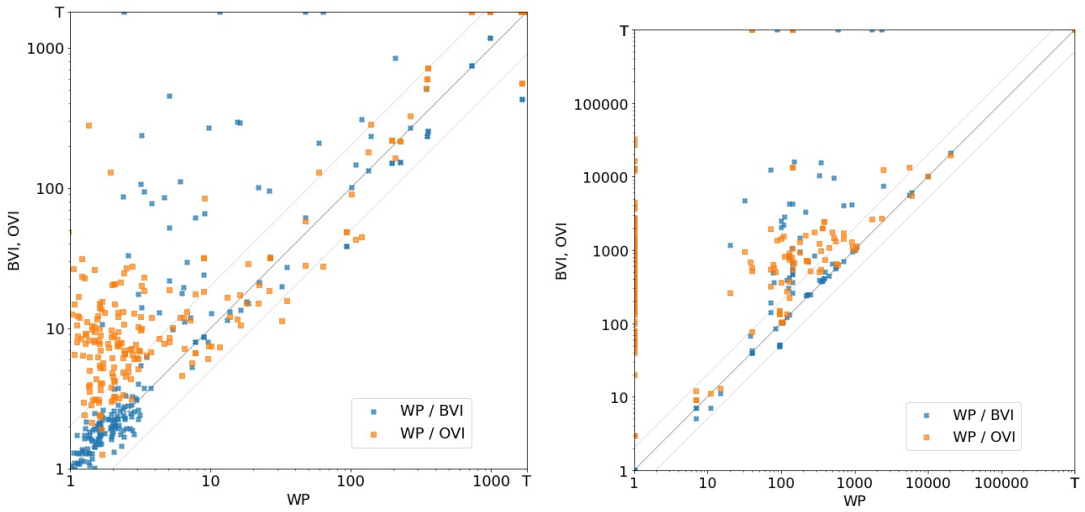
On one hand, Figure 7.4(a) indicates that $\mathbf{BVI^{PT}}$ is the least performant algorithm, while $\mathbf{BVI^{D}}$, $\mathbf{SI_{SI}}$, and $\mathbf{SI_{VI}}$ perform decently. On the other hand, in Figure 7.4(b) $\mathbf{BVI^{PT}}$ is significantly better, while $\mathbf{BVI^{D}}$, $\mathbf{SI_{SI}}$, and $\mathbf{SI_{VI}}$ are significantly worse than the other algorithms. However, for both the real case studies and the randomly generated models, **WP** and $\mathbf{OVI^{G}}$ are the most performant

algorithms. Furthermore, for both sets of models, the topological optimization **OVI**$^{\mathbf{T}}$ fails to solve significantly more models correctly than **OVI** or **OVI**$^{\mathbf{G}}$.

We split our algorithm analysis into three subtopics: First, we compare the three value-iteration-based algorithms with guarantees **OVI**, **WP**, and **BVI**. Then, we analyze the impact of the optimizations from Subsection 2.7.4 on their respective baseline algorithms. Lastly, we investigate the performance of strategy-iteration-based algorithms in comparison to value-iteration-based algorithms. For **OVI**, **WP**, **BVI**, and their optimizations, we compare the number of iterations required to solve a model in addition to the time. While time is practically more relevant, the number of iterations is independent of how sophisticated the implementation of the algorithm is.

7.4.1. BVI vs OVI vs WP

First, we compare bounded value iteration (**BVI**), optimistic value iteration (**OVI**), and widest path bounded value iteration (**WP**) regarding their performance. We observe in Figure 7.4 that **WP** is the fastest approach for solving both randomly generated models and real case studies. To see whether this is the case for all models or only when accumulating runtime, we provide a scatter plot in Figure 7.5. The x-axis marks the time/iterations **WP** requires to solve a stochastic game, and the y-axis marks the respective time/iterations **BVI** or **OVI** needs. The two thin lines next to the diagonal mark the case that **WP** is twice as fast as **BVI / OVI** or half as fast.



(a) Time required to solve a model

(b) Iterations required to solve a model

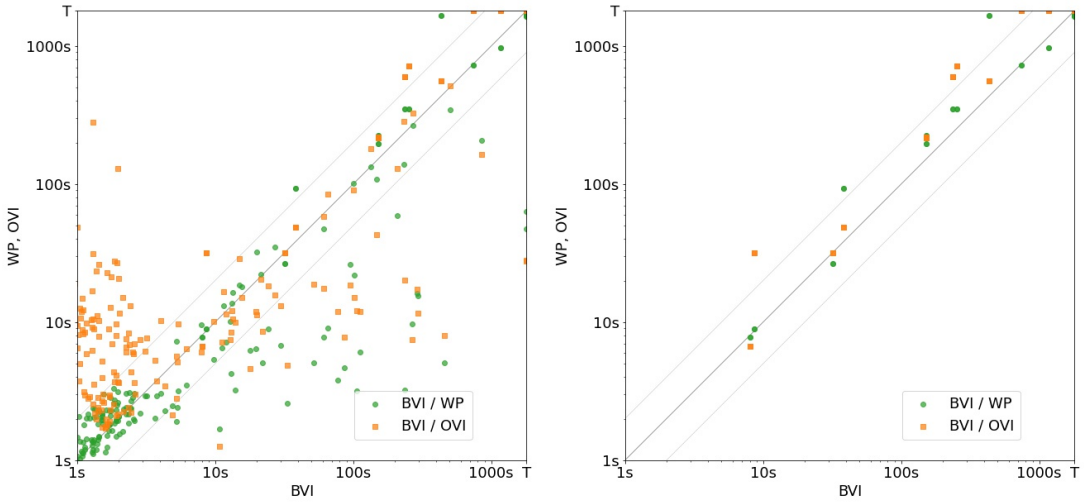
Figure 7.5.: **WP** compared to **BVI** and **OVI** on every model of datasets (i), (ii), and (iii) from Subsection 7.1.1. See 7.1.2 for a reference on how to read the plot.

WP

Regarding both time and iterations, although **WP** is the fastest algorithm when measuring accumulated runtime, it is neither consistently better nor worse than **OVI** and **BVI** on our benchmarks. However, note that the difference between **BVI** and **WP** is how they solve end components. In our models, there are usually less than 3 MECs, and their size usually does not exceed 5 states. Thus, we have created a dataset randomly using the RandomSCC guideline with 10 models and 10000 states each. Each model contains a large MEC with at least 9000 states. While **WP** required per model no more than 30 seconds, **OVI** solved only 3 models with 5 to 9 minutes per model, and **BVI** solved none within a 15 minutes time frame. Since **WP** is consistently one of the fastest algorithms and excels at solving MECs, we conclude that it is the best initial choice in case of doubt.

BVI

Figure 7.6 provides an overview of the time **BVI** requires to solve a model compared to **OVI** and **WP**. While both the accumulated runtime of Figure 7.4 and Figure 7.6(a) suggest that for more complicated models, **OVI** and **WP** are faster than **BVI**, Figure 7.6(b) shows that for large models, **BVI** tends to be faster. However, we emphasize that this tendency may be due to a lack of a diverse dataset for large models.



(a) Time required to solve for all models from Subsection 7.1.1

(b) Time required to solve for all large models as described in Subsection 7.1.1 (iv)

Figure 7.6.: **BVI** compared to **WP** and **OVI**. See 7.1.2 for a reference on how to read the plot.

We found that **BVI** usually requires fewer iterations than **OVI** to solve a model. The reasons

for that are explained in the following paragraph:

OVI

When considering models without large MECs, **OVI** is faster than **BVI** as Figure 7.6 indicated, but not significantly worse or better than **WP** for most models as shown in Figure 7.5. However, optimistic value iteration is a heuristic whose performance depends on never failing when guessing a bound. We recall that whenever **OVI** guesses a bound that is neither an upper nor a lower bound, it halves its ε' and must continue iterating on the lower bound. The next verification phase will only start once no states value changes by more than the new ε'. Thus, the verification phase may fail even though the value of each state is already very close to the required precision. In that case, **OVI** will perform unnecessary extra iterations until it reaches ε' and verifies that **OVI** may indeed terminate.

We found that in every case where **OVI** failed one verification phase, it would not finish computation within the given time frame. However, Figure 7.7 also provides evidence that optimistic value iteration almost never fails in a verification phase. Note that all models in the left upper quadrant (**OVI** requires at least 100s, **WP** less than 100s) are from the set with large MECs mentioned in the paragraph about **WP** (Subsection 7.4.1). Since **BVI** times out on every of these models, the bad performance of **OVI** is probably due to deflating the end components. Furthermore, for models that are hard for value-iteration-based algorithms like the adversarial model from [HM18] **OVI** may reach the time limit before failing one verification phase.

On the other hand, there are also conditions for which **OVI** is better than **WP** and **BVI**. While both **WP** and **BVI** require that both lower and upper bound converge, **OVI** can solve models easily where the lower bound converges quickly, but the upper bound does not. However, we were unable to successfully correlate and model feature we track to fast lower bound convergence paired with slow upper bound convergence.

Next, we evaluate the number of iterations optimistic value iteration needs. **OVI** generally requires more iterations than **WP** and **BVI**. This is partly because of the extra iterations that may happen in **OVI** as mentioned, and partly because during the verification phase only the upper bound is being iterated on. Meanwhile, in **BVI** both upper- and lower bound are updated per iteration. Also, it is interesting to see that for about half of the models in Figure 7.5, **WP** requires only 1 iteration whereas **OVI** requires many iterations to solve the model. The red dots in Figure 7.8 visualize the value of the games where **WP** required only 1 iteration and **OVI** multiple, while the green dots represent models where **WP** required also multiple iterations to solve the game.

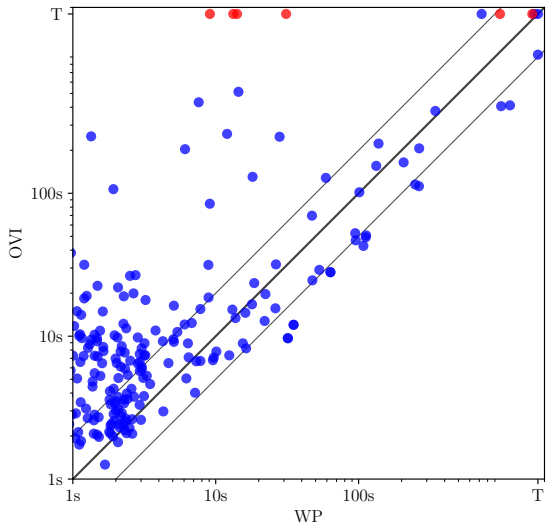


Figure 7.7.: A scatter plot comparing **WP** to **OVI** on all models sets including the 10 models where each model has a MEC with at least 9000 states. See 7.1.2 for a reference on how to read the plot. The red dots visualize models where **OVI** failed a verification phase, while in the blue dots there was no failed verification phase.

Figure 7.8.: A one dimensional scatter plot of when **WP** could solve a model in 1 iteration while **OVI** could not (red) against where **WP** requires more than 1 iteration (green).

Clearly, in some instances where a game has value 0 or 1, **WP** is able to identify its value in only 1 iteration while **OVI** cannot. **BVI** is also capable of solving these models in 1 iteration. In the runtime scatter plot of Figure 7.5, these models are mostly the cloud of orange dots in the lower left quadrant. However, even if **BVI** and **WP** require only one iteration, they still require various seconds to perform this iteration, as Figure 7.9. The outliers that require more than 80 seconds to solve for **BVI** are variations of the real case study "Avoid the Observer" from [Cha+20].

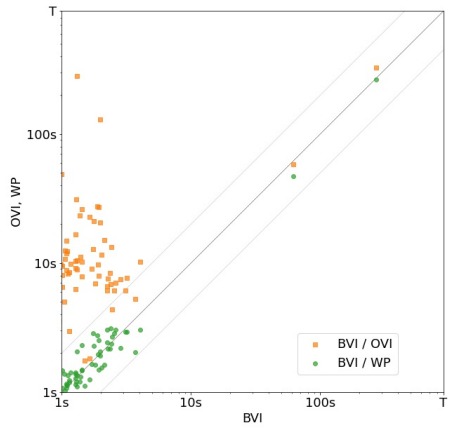


Figure 7.9.: A scatter plot of the time **BVI** requires solving models where **BVI** and **WP** require only one iteration to solve the model, but **OVI** requires multiple iterations.

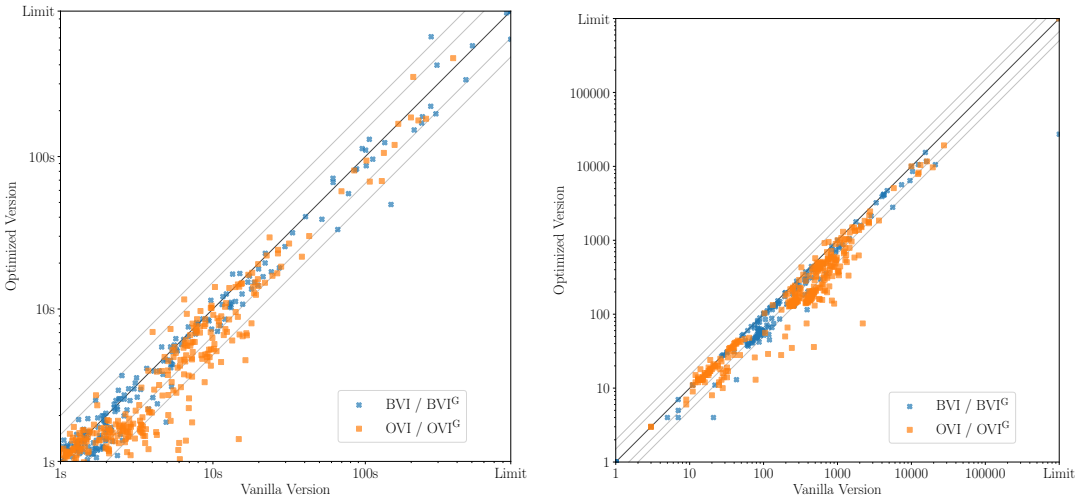
7.4.2. Analyzing the optimizations

After comparing the extensions for value iteration, we evaluate the impact of including optimizations like topological ordering or using the Gauss-Seidel optimization. Figure 7.4 indicates that optimizations introduced in Subsection 2.7.4 do not always improve the accumulated runtime. For example, while $\mathbf{BVI^D}$ is better than **BVI** in the real case studies, it is significantly worse on our randomly generated models. To analyze the effect of the optimizations, we compare **BVI**, **OVI**, and $\mathbf{SI_{LP}}$ with the optimizations that apply to them in scatter plots. For each optimization, we provide a scatter plot with the time required to solve the models, and one scatter plot with iterations required to solve the models. For the iterations scatter plots, we include only **BVI** and **OVI** since iterations in strategy iteration are not comparable to iterations in value iteration.

Gauss – Seidel for BVI:

As Figure 7.10 suggests, using the Gauss-Seidel optimization may reduce both the time and iterations required to solve a model by up to 4 times in most cases. The Gauss-Seidel optimization is slower in some cases because the values are computed state by state to enable using already computed results. The unoptimized version uses vector operations instead, which is faster sometimes.

In a few instances, the Gauss-Seidel optimization requires more iterations than the unoptimized version. This is because **OVI** and **BVI** may find different end components to deflate depending on whether Gauss-Seidel is used or not. In some cases, the unoptimized version is able to find more favorable sets of end components and requires thus fewer iterations. Changing the order of computation of the states for the Gauss-Seidel optimization by computing along a topological



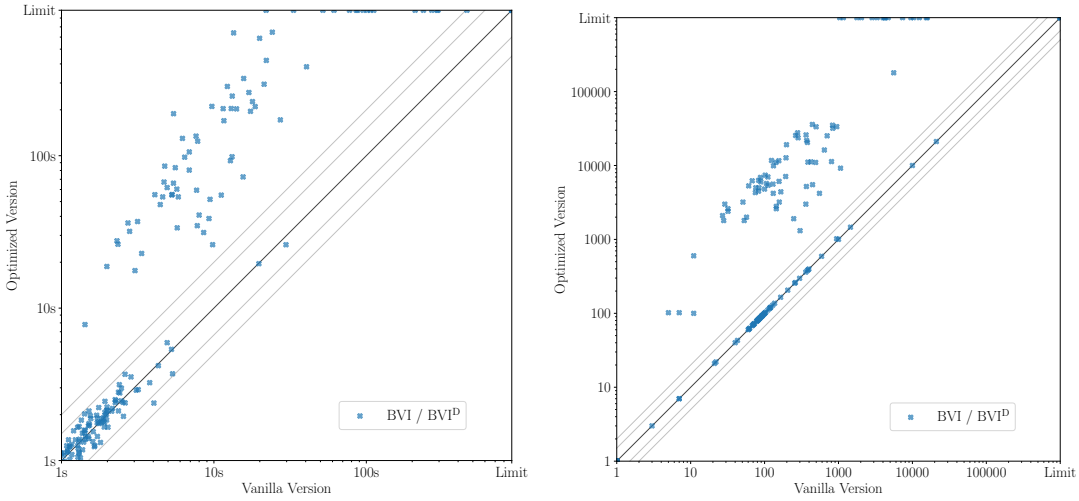
(a) Time required to solve a model

(b) Iterations required to solve a model

Figure 7.10.: **BVI** and **OVI** compared to their Gauss-Seidel optimizations on every model of datasets (i), (ii), and (iii) from Subsection 7.1.1

enumeration of the states did not yield improvements in our experiments.

D for BVI:



(a) Time required to solve a model

(b) Iterations required to solve a model

Figure 7.11.: **BVI** compared to its optimization where deflating happens only every 100 iterations on every model of datasets (i), (ii), and (iii) from Subsection 7.1.1

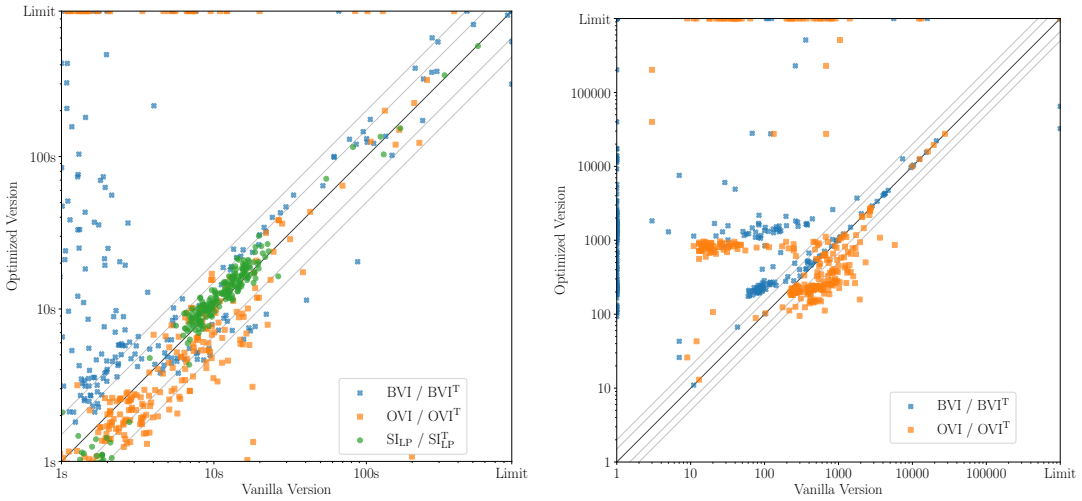
Figure 7.11 clearly indicates that although $\mathbf{BVI^D}$ solves some models faster, for most of our models, it could not compete with **BVI**. Almost all models where **BVI** is more than twice as fast as $\mathbf{BVI^D}$ have at least three MECs. Furthermore, almost all models where **BVI** and $\mathbf{BVI^D}$ require the same number of iterations have no end components.

T for BVI, OVI and $\mathbf{SI_{LP}}$:

As Figure 7.12 displays, the topological optimization improves the runtime of **OVI** considerably. However, $\mathbf{OVI^T}$ returns an incorrect value in around 10% of all models.

From the strategy iteration variants, we use $\mathbf{SI_{LP}}$ since it is the most performant variant. For $\mathbf{SI_{LP}}$, the topological optimization does neither in nor decrease the performance of the algorithm considerably except for case studies like dice or MulMEC that have long chains of SCCs, and favor topological algorithms. Since $\mathbf{SI_{LP}}^{\mathbf{T}}$ incurs very little overhead, we conclude that if in doubt, use the topological variant.

Lastly, the topological optimization affects **BVI** usually negatively. As explained in [Kře+20], chains of subsequent SCCs are harder to compute for $\mathbf{BVI^T}$ because the values of the already computed SCCs are not exact, making it harder for the following SCCs to reach ε-precision. Another reason why **BVI** may be faster than its topological extension is that in one iteration,



(a) Time required to solve a model

(b) Iterations required to solve a model

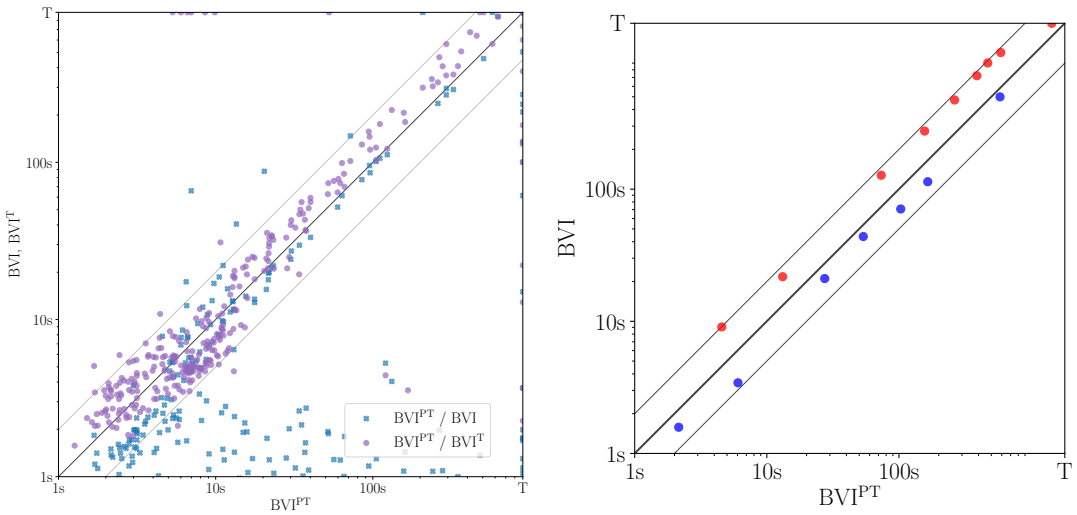
Figure 7.12.: **BVI**, **OVI** and $\mathbf{SI_{LP}}$ compared to their topological optimizations on every model of datasets (i), (ii), and (iii) from Subsection 7.1.1

every state in the whole graph is updated, while in the topological extension, only states in the current SCC are updated. An extreme example is a tree-like game with many states but only two levels: The initial state is the root and has a transition to every state, and every state is an SCC. While the topological extension will solve every other SCC before solving the initial state, standard **BVI** may converge within significantly fewer iterations. Note that this is not specific to $\mathbf{BVI^T}$ but any topological approach. However, due to time limitations, we were unable to test this hypothesis.

$\underline{\mathbf{BVI^{PT}}}$:

$\mathbf{BVI^{PT}}$ was introduced to solve the issues $\mathbf{BVI^T}$ has with long chains of SCCs. It successfully tackles the issues of $\mathbf{BVI^T}$ and solves adversarial models for $\mathbf{BVI^T}$ like the handcrafted model MulMEC fast. While it is not generally better than $\mathbf{BVI^T}$ on our benchmarks as Figure 7.13(a) indicates, solving multiple SCCs is the only case where topological optimizations should be used. Thus, we conclude that $\mathbf{BVI^{PT}}$ makes $\mathbf{BVI^T}$ obsolete.

When comparing $\mathbf{BVI^{PT}}$ to **BVI**, it is clear that $\mathbf{BVI^{PT}}$ - like the topological optimization - impacts **BVI** usually negatively. $\mathbf{BVI^{PT}}$ is slower than **BVI** at solving most models for whichs $\mathbf{BVI^T}$ is also slower than **BVI**. Again, this might be due to the fact that **BVI** updates the whole state-space while topological variants must solve every SCC consecutively. However, given a model that consists contains chains of large SCCs, $\mathbf{BVI^{PT}}$ is faster than standard bounded value



(a) Time $\mathbf{BVI^{PT}}$ requires to solve a model of the datasets (i), (ii), and (iii) in comparison to **BVI** and $\mathbf{BVI^{T}}$.

(b) Time $\mathbf{BVI^{PT}}$ requires to solve a model of the dataset (iv) in comparison to **BVI**.

Figure 7.13.: Two scatter plots of the time $\mathbf{BVI^{PT}}$ requires to solve a model in comparison to both **BVI** and $\mathbf{BVI^{T}}$. For scatter plot (a), we have used real (i), handcrafted (ii) models, and randomly generated models (iii) as described in Subsection 7.1.1. For scatter plot (b), we used the large, tree-like models (iv) as described in Subsection 7.1.1. In the scatter plot (b), the blue dots are models where the whole state space consists of one large SCC, while models marked red dots have their state space divided into five equally large SCCs.

iteration. Figure 7.13(b) indicates that if the model consists of one SCC, **BVI** is better than $\mathbf{BVI^{PT}}$ if there is only one SCC, but when the state space is divided into five equally sized SCCs, $\mathbf{BVI^{PT}}$ is more performant.

Lastly, note that $\mathbf{BVI^{PT}}$ may infer wrong strategies when computing the precise value for an SCC. In that case, $\mathbf{BVI^{PT}}$ throws an error and terminates. On our benchmark, in 5 out of 350 models this error was thrown.

7.4.3. Algorithms based on strategy iteration

Although value iteration was regarded for a long time to be the most performant algorithm type for solving stochastic games, [Kře+20] showed that $\mathbf{SI_{VI}}$ is a valid option, and that mathematical programming can be good sometimes. Thus, in addition to $\mathbf{SI_{VI}}$, we benchmark $\mathbf{SI_{SI}}$ and $\mathbf{SI_{LP}}$, which do not use value iteration to solve stochastic games.

Strategy iteration with linear programming

On most of our models, $\mathbf{SI_{LP}}$ yielded the best results alongside widest path bounded value iteration. Since strategy iteration simply tries to make an informed decision on which strategy to pick and solves the underlying MDP, we have to inspect the algorithms we use to solve MDPs - for $\mathbf{SI_{LP}}$ this is linear programming.

Linear programming scales worse than value iteration for huge models. On the large models used in Figure 7.13(b), $\mathbf{SI_{LP}}^{\mathbf{T}}$ is significantly less performant than $\mathbf{BVI}^{\mathbf{PT}}$, independent of whether the model consisted of one or five SCCs. If an SCC has at least 2 million states, $\mathbf{SI_{LP}}^{\mathbf{T}}$ could not solve the corresponding model while $\mathbf{BVI}^{\mathbf{PT}}$ could. Additionally, as in [Kře+20], we found that mathematical programming requires more memory than value iteration and thus is more prone to encounter out-of-memory exceptions for large models. In preliminary experiments, we found that SCCs with more than 10 million states would run out of memory when using linear programming for MDPs and applying a memory limit of 36 GB. Thus, we do not recommend using either $\mathbf{SI_{LP}}$ or $\mathbf{SI_{LP}}^{\mathbf{T}}$ on models with numerous states.

However, $\mathbf{SI_{LP}}^{\mathbf{T}}$ may be a good complementary solution approach in case a model is especially hard for value-iteration-based approaches. The adversarial value iteration model hm from [HM18] is solved by strategy iteration with linear programming within less than a second.

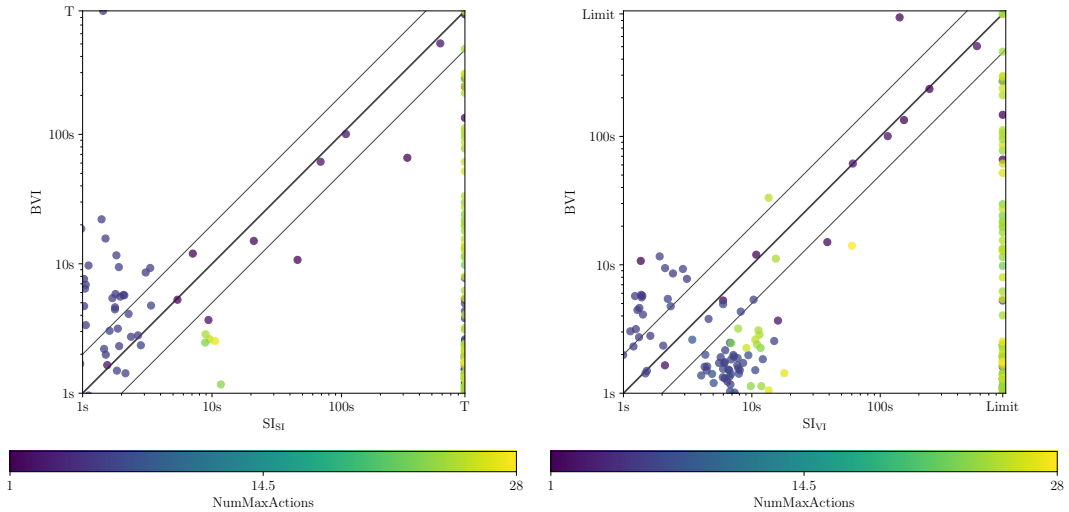
Lastly, in contrast to the other strategy iteration variants $\mathbf{SI_{SI}}$ and $\mathbf{SI_{VI}}$, we could not find solid evidence that $\mathbf{SI_{LP}}$ is negatively affected by increasing numbers of actions in a model.

Strategy iteration with Markov chain solving

While $\mathbf{SI_{SI}}$ was unable to solve four out of 18 models in the real case studies, it failed at solving 179 out of 284 randomly generated models. Figure 7.14(a) indicates that this variant of strategy iteration is unable to solve models that have states with large numbers of actions. This algorithm has to solve simple Markov chains instead of MDPs like $\mathbf{SI_{LP}}$, and thus is less prone to out-of-memory errors. However, it has to solve significantly more linear equation systems. We found that for the benchmark (iv) with large models, $\mathbf{SI_{SI}}$ timed out whenever a model had more than 500000 states.

Strategy iteration with value iteration

While we confirm the findings of [Kře+20] that for the real models $\mathbf{SI_{VI}}$ is comparable with **BVI**, the accumulated runtime for the randomly generated models displayed in Figure 7.4(b) clearly shows that the models generated we generated are highly unfavorable for $\mathbf{SI_{VI}}$.



(a) $\mathbf{SI_{SI}}$ compared to **BVI**

(b) $\mathbf{SI_{VI}}$ compared to **BVI**

Figure 7.14.: A scatter plot of the time that (a) $\mathbf{SI_{SI}}$ and (b) $\mathbf{SI_{VI}}$ require to solve a model from the datasets (i), (ii), (iii) as described in Subsection 7.1.1 against the time **BVI** required. The dots are colored by the maximal number of actions per state. The more the color of a point goes towards yellow, the higher the maximal number of actions a state in the corresponding model has.

This is most likely due to the higher number of actions per state of our generated models compared to the case studies. As Figure 7.14(b) suggests, if every state has up to 3 actions, $\mathbf{SI_{VI}}$ and **BVI** perform similarly. However, as soon as there are states with more actions, $\mathbf{SI_{VI}}$ is performing worse. Another problem of strategy iteration with value iteration for MDPs is that, compared to $\mathbf{SI_{LP}}$ for example, adversarial models for value iteration are also hard for $\mathbf{SI_{VI}}$. In addition to the approximately 100 models $\mathbf{SI_{VI}}$ did not solve in time, 5 models out of the 250 solved did not yield a correct result since using value iteration for MDP-solving does not guarantee correct results.

8. Conclusion and future work

In this thesis, we introduced a toolset that randomly generates models to enrich the structural variety of available models. This enabled a broader and more precise comparison of the available solution algorithms.

We have shown that our approach to generating models at random can create both data sets that share biases with the available real case studies, and data sets where the structural properties differ significantly from the case studies. However, although we can generate models of arbitrary size, we are limited by the slow state exploration process of the PRISM model checker.

Furthermore, establishing analysis tools allowed us to prototype algorithm ideas, compare algorithm performance, and provide experimental evidence of a correlation between model structure and algorithm. Additionally, our tools for model analysis improve with increasing numbers of data points. Although this is not the case for tables as visualization and analysis tools, they are nevertheless frequently the only utility used for visualization in analysis [Kel+18; Pha+20; Brá+14].

Lastly, we have compared algorithms that solve SG. We found that, in general, **WP** is faster than every other algorithm that provides a guarantee on the precision of the value. In our benchmarks, we could not find any class of models where **BVI** or **OVI** are consistently more performant than **WP**. **WP** is significantly better than every other algorithm at solving models with few large MECs that contain almost the whole state space. Thus, we conclude that in case of doubt, **WP** is the initial algorithm we recommend.

Furthermore, we found that if a models state space is divided into a chain of SCCs, $\mathbf{BVI}^{\mathbf{PT}}$ is usually the fastest algorithm to solve the problem. We expect topological algorithms to be less performant the shorter the chains of SCCs are.

We showed that the best algorithm based on strategy iteration is $\mathbf{SI_{LP}}^{\mathbf{T}}$. While linear programming is inefficient for large models, it may be a good alternative for games that are adversarial to approaches based on value iteration. In case **WP** and $\mathbf{BVI}^{\mathbf{PT}}$ fail, we suggest using this algorithm.

Future work

Although we have provided the first steps towards improving algorithm analysis for SGs, there are still many fields of our work that act as a base for further improvements.

The model features we analyze at the moment helped us to draw some conclusions, but there are still many questions unanswered that would require additional structural concepts and features

that we track. These questions are, for example, when to use Gauss-Seidel optimization or whether there is a model structure that consistently makes **OVI** faster than **WP**.

In addition, there are still many data analysis techniques we did not implement yet, like stochastic tests or artificial intelligence algorithms that can cluster features and find correlations between them.

We believe that more research on the correlation between algorithm performance and structural properties gives way to a portfolio solver, which could first analyze important structural features, and then pick heuristically the most adequate algorithm to solve the problem. With enough models, the decision of which algorithm to use may be made by a neural network. Also, combinations of algorithms like **OVI** and **WP** could be used effectively in a fitting scenario. The user would have to pay the overhead for computing more vectors but may converge faster.

Since algorithm performance may change drastically depending on the models' size, we need to resolve the issue that our random generation process creates large explicit files which PRISM cannot handle. The next step would be to create an optimization system for our random generation algorithm that takes advantage of the PRISM language and holds chunks of the data in implicit blocks or uses modules to make the files easier parsable.

Furthermore, at the moment, our random generation algorithm enforces a tendency of states with smaller indices to have more actions than states with higher indices. Ideally, the user should be able to remove this restriction. This could be addressed by introducing mechanisms like, for example, that a state s_i can only be connected to states s_j with $j \in [i-50, i+50]$. Another way to improve the random generation algorithm is to find a method of reducing the models' numbers of states that can be computed trivially.

Ultimately, improving random algorithm generation and learning more about model structure is the basis for what could become a uniform benchmarking set that could be used by everyone creating algorithms for SGs to test and improve their algorithms.

In conclusion, we have taken the first step towards improving algorithm analysis for SGs, and we have provided many promising directions for improvements of the algorithm analysis, the random model generation, and the algorithms themselves.

List of Figures

Bibliography

[Aze+ed] M. Azeem, A. Evangelidis, J. Křetínský, A. Slivinskiy, and M. Weininger. "Optimistic and Topological Value Iteration for Simple Stochastic Games." In: *CAV 2022*. Lecture Notes in Computer Science. Springer, submitted.

[Bal+19] N. Balaji, S. Kiefer, P. Novotný, G. A. Pérez, and M. Shirmohammadi. "On the Complexity of Value Iteration." In: *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Ed. by C. Baier, I. Chatzigiannakis, P. Flocchini, and S. Leonardi. Vol. 132. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 102:1–102:15. ISBN: 978-3-95977-109-2. DOI: 10.4230/LIPIcs.ICALP.2019.102.

[BK08] C. Baier and J.-P. Katoen. *Principles of model checking*. 2008.

[Brá+14] T. Brázdil, K. Chatterjee, M. Chmelík, V. Forejt, J. Křetínský, M. Kwiatkowska, D. Parker, and M. Ujma. "Verification of Markov Decision Processes using Learning Algorithms." In: *Proc. 12th International Symposium on Automated Technology for Verification and Analysis (ATVA'14)*. Vol. 8837. LNCS. Springer, 2014, pp. 98–114.

[CdH13] K. Chatterjee, L. de Alfaro, and T. A. Henzinger. "Strategy improvement for concurrent reachability and turn-based stochastic safety games." In: *Journal of Computer and System Sciences* 79.5 (2013), pp. 640–657. ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2012.12.001.

[CH08] K. Chatterjee and T. A. Henzinger. "Value Iteration." In: *25 Years of Model Checking*. Vol. 5000. Lecture Notes in Computer Science. Springer, 2008, pp. 107–138. DOI: 10.1007/978-3-540-69850-0_7.

[Cha+10] K. Chatterjee, T. A. Henzinger, B. Jobstmann, and A. Radhakrishna. *GIST: A Solver for Probabilistic Games*. 2010. arXiv: 1004.2367 [cs.LO].

[Cha+20] K. Chatterjee, J.-P. Katoen, M. Weininger, and T. Winkler. "Stochastic Games with Lexicographic Reachability-Safety Objectives." In: vol. abs/2005.04018. 2020. arXiv: 2005.04018.

[Che+11a] T. Chen, M. Kwiatkowska, D. Parker, and A. Simaitis. "Verifying Team Formation Protocols with Probabilistic Model Checking." In: *Proc. 12th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA XII 2011)*. Vol. 6814. LNCS. Springer, 2011, pp. 190–297.

[Che+11b] C.-H. Cheng, A. Knoll, M. Luttenberger, and C. Buckl. "GAVS+: An Open Platform for the Research of Algorithmic Game Solving." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by P. A. Abdulla and K. R. M. Leino. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 258–261. ISBN: 978-3-642-19835-9.

[Che+13] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. "PRISM-games: A Model Checker for Stochastic Multi-Player Games." In: *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Ed. by N. Piterman and S. Smolka. Vol. 7795. LNCS. Springer, 2013, pp. 185–191.

[Con92] A. Condon. "The Complexity of Stochastic Games." In: *Inf. Comput.* 96.2 (1992), pp. 203–224. DOI: 10.1016/0890-5401(92)90048-K.

[Con93] A. Condon. "On Algorithms for Simple Stochastic Games." In: (1993), pp. 51–73.

[CY95] C. Courcoubetis and M. Yannakakis. "The Complexity of Probabilistic Verification." In: *J. ACM* 42.4 (1995), pp. 857–907. DOI: 10.1145/210332.210339.

[GS06] C. M. Grinstead and J. L. Snell. "Introduction to Probability." In: second revised edition. American Mathematical Society, 2006. Chap. 11.

[Hah+19] E. Hahn, A. Hartmanns, C. Hensel, M. Klauck, J. Klein, J. Křetínský, D. Parker, T. Quatmann, E. Ruijters, and M. Steinmetz. "The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models: (QComp 2019 Competition Report)." In: Apr. 2019, pp. 69–92. ISBN: 978-1-4939-9100-6. DOI: 10.1007/978-3-030-17502-3_5.

[HK19] A. Hartmanns and B. L. Kaminski. "Optimistic Value Iteration." In: *CoRR* abs/1910.01100 (2019). arXiv: 1910.01100.

[HK66] A. J. Hoffman and R. M. Karp. "On Nonterminating Stochastic Games." In: *Management Science* 12.5 (1966), pp. 359–370. ISSN: 00251909, 15265501.

[HM18] S. Haddad and B. Monmege. "Interval iteration algorithm for MDPs and IMDPs." In: *Theor. Comput. Sci.* 735 (2018), pp. 111–131. DOI: 10.1016/j.tcs.2016.12.003.

[Kel+18] E. Kelmendi, J. Krämer, J. Křetínský, and M. Weininger. "Value Iteration for Simple Stochastic Games: Stopping Criterion and Learning Algorithm." In: (2018). Ed. by H. Chockler and G. Weissenbacher, pp. 623–642.

[KNP12] M. Z. Kwiatkowska, G. Norman, and D. Parker. "The PRISM Benchmark Suite." In: *Ninth International Conference on Quantitative Evaluation of Systems, QEST 2012, London, United Kingdom, September 17-20, 2012*. IEEE Computer Society, 2012, pp. 203–204. DOI: 10.1109/QEST.2012.14.

[Kře+20] J. Křetínský, E. Ramneantu, A. Slivinskiy, and M. Weininger. "Comparison of Algorithms for Simple Stochastic Games (Full Version)." In: *CoRR* abs/2008.09465 (2020). arXiv: 2008.09465.

[Kwi+20] M. Kwiatkowska, G. Norman, D. Parker, and G. Santos. "PRISM-games 3.0: Stochastic Game Verification with Concurrency, Equilibria and Time." In: *CAV (2)*. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 475–487. DOI: 10.1007/978-3-030-53291-8_25.

[Pha+20] K. Phalakarn, T. Takisaka, T. Haas, and I. Hasuo. "Widest Paths and Global Propagation in Bounded Value Iteration for Stochastic Games." In: *CoRR* abs/2007.07421 (2020). arXiv: 2007.07421.

[Put14] M. L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[Sha53] L. S. Shapley. "Stochastic games." In: *Proceedings of the national academy of sciences* 39.10 (1953), pp. 1095–1100.

[Tar72] R. Tarjan. "Depth first search and linear graph algorithms." In: *SIAM JOURNAL ON COMPUTING* 1.2 (1972). DOI: 10.1137/0201010.

A. Appendix: Information about the models

Random model generation parameters

In this section, we present the parameters we have used to generate the 300 randomly generated models. We have generated three sets of random models: Two for Algorithm 1 with a distinct set of parameters, referred to as random and RandomB, and one with the RandomSCC guideline. While we have also tried and used the RandomTree guideline, we did not include it in our benchmarks due to a lack of time. All randomly generated models can be found in the GitHub Repository `https://github.com/ga67vib/Algorithms-For-Stochastic-Games` in the folder `random-generated-models`. The difference between the RandomA and RandomB is that we chose the parameters for RandomA so that the dataset resembles the biases of the real case studies, while we chose the parameters for RandomB such that every model has many actions. The feature distributions of RandomA are displayed in Figure 7.2, and the feature distributions of the set created with the RandomSCC guideline are displayed in Figure 7.3. The parameters we have used to generate these models were the following:

Parameter	RandomA	RandomB	RandomSCC
size	10000	10000	10000
numModels	100	100	100
guideline	-	-	RandomSCC
smallestProb	10^{-2}	10^{-3}	10^{-3}
backwardsProb	1	1	1
maxBranchNum	10	10	10
forceUnknown	false	false	true
branchingProb	1	0.75	0.75
maximizerProb	0.5	0.5	0.7
minIncomingActions	2	10	10
maxIncomingActions	4	12	12
maxBackwardsActions	1	1	10
minSCCSize	-	-	1000
maxSCCSize	-	-	2000