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

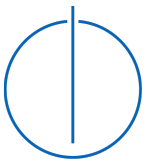
TECHNISCHE UNIVERSITÄT MÜNCHEN

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:	28.01.2022



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

Munich, 28.01.2022 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 stochastic settings with reactive systems. 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. While there are many algorithms to solve stochastic games with reachability objective, there is a lack of experimental data that can be used as a baseline for an informed decision on which algorithm to use in which situation. 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 game, 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 dealing with this question. 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.

However, in the real world, some events only happen occasionally, like bad weather while driving somewhere or that a part in factory breaks. Thus, we need to quantify how often these realistic events occur and make our theoretical model probabilistic. Through *probabilistic model checking* it is then possible to 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 how high the probability is that the first player reaches one of these goal states. Through this, we can derive what we wanted to 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].

To 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 NUMBER

OF MODELS distinct models. With CITATIONS TO OTHER MODELS 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 a bad 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 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 the algorithm performance.

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 in general 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 contains 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 solving simple stochastic games has a complexity in **NP** $\cap$ 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 Citation as commentRecently, several heuristics were introduced that provide a guarantee on how close value iteration is away from the result for MDPsCopy from Maxi and SGs [Kel+18]. More stuff about VI

more stuff about SI

[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.

The tools implementing the standard SI and/or VI algorithms for SGs are PRISM-games [Che+13a], GAVS+ [Che+11] and GIST [Cha+10]. GAVS+ is however not maintained anymore, and 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 in Section 2.1 an intuition. 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. Lastly, we give 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 $\mathsf{S}_{\square}$ 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 $s \in \mathsf{S}_{\square}$ 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?*

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

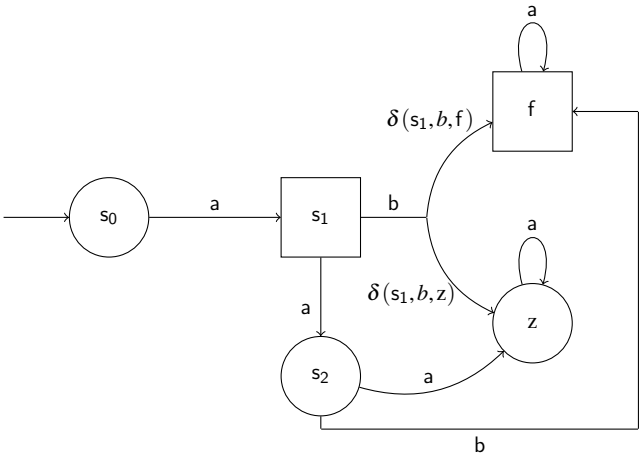


Figure 2.1.: An example of a simple stochastic game. $s_0, s_1, s_2, \mathsf{f}, \mathsf{z}$ are states. a, b are actions. s_1 and f are Maximizer-states, while the rest of the states belongs to the Minimizer. f is a target and z is a so-called sink, i.e. a state that has no path to the target. $\delta(\mathsf{s}_1, b, \mathsf{f})$ and $\delta(\mathsf{s}_1, b, \mathsf{z})$ are the transition probabilities that a token in s_1 moved through action b would either be moved to f or z

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* $(\mathsf{S}, \mathsf{S}_\square, \mathsf{S}_\circ, \mathsf{s}_0, \mathsf{A}, \mathsf{Av}, \delta)$ *where* S *is a finite set of* states *partitioned*[1] *into the sets* $\mathsf{S}_\square$ *and* $\mathsf{S}_\circ$ *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* $\mathsf{S}_\circ = \emptyset$ *or* $\mathsf{S}_\square = \emptyset$*, 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 as $\mathsf{Post}(\mathsf{s}, \mathsf{a}) := \{\mathsf{s}' \mid \delta(\mathsf{s}, \mathsf{a}, \mathsf{s}') > 0\}$. 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 $\mathsf{Av}(\mathsf{s}) \neq \emptyset$.

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

2.3. Strategies

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

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: $\forall \mathsf{s} \in \mathsf{S}_{\square} : \exists \mathsf{a} \in \mathsf{Av}(\mathsf{s}) : \sigma(\mathsf{s},\mathsf{a}) = 1$ and $\forall \mathsf{s} \in \mathsf{S}_{\circ} : \exists \mathsf{a} \in \mathsf{Av}(\mathsf{s}) : \tau(\mathsf{s},\mathsf{a}) = 1$. 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 an SG with one target f and one sink z.

Intuitively, we can 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 avoid 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 can 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

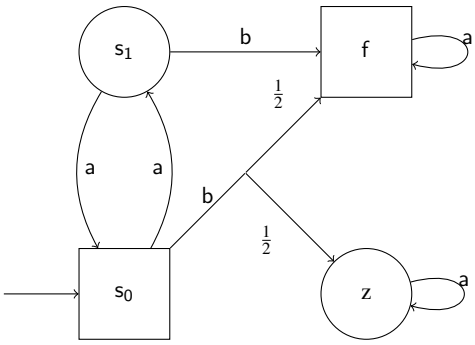


Figure 2.2.: An example of a non-stopping SG with an end component of size 2. If in s_0 action a and in s_1 action a are chosen for some strategies σ, τ the game does never stop. $\mathsf{V}_{\sigma,\tau}(s_0) = 0$ because the probability to reach any $f \in F$ is 0 with these strategies.

optimal 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 this 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 $\delta(s_1, b, \mathsf{f})$. This is because s_0 has to pick action a leading into s_1. For the Maximizer, action b is better than action a as long as $\delta(s_1, b, \mathsf{f}) > 0$, since action a in state s_1 leads into s_2, which in turn would always pick action a leading always into the z.

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 SG based on its SCCs and solve SG with a divide-and-conquer approach. Instead of computing the values of the states in the stochastic game, one can instead compute 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 can *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* that do not have a path to the target Z 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 this approach 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 greater fixpoint

[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.

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

Optimistic value iteration **OVI** takes a similar approach, but instead of calculating the upper bound at all times along the lower bound, an upper bound B is guessed from the lower bound by adding a small ε to the lower bound. If this induced bound B can be verified to be an upper bound, i.e. $\forall \mathsf{s} \in \mathsf{S} : \mathcal{B}(B)(\mathsf{s}) \leq B(\mathsf{s})$, then **OVI** terminates. The upper bound is only guessed if the lower bound would converge according to classic value iteration. If the guessed bound is not an upper bound, the precision for the lower bound gets reduced, and iterating continues.

There are more extensions to value iteration 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 [HM18], 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 with an optimal strategy for both players. Starting from an arbitrary strategy for the Maximizer, we repeatedly compute the best response of Minimizer and then greedily improve Maximizer's strategy. This is word for word copied from gandalf, so I should cite it? 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, which we solve with classic exact algorithms as described in [GS06, Chapter 11].

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 can confirm 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 can use already the 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 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.

PT: A novel variant of the topological optimization for value iteration that is unpublished at this moment How to cite to it if unpublished?. To ensure that each SCC passes the correct value onto the next SCC, we use the computed value to infer strategies for the states and verify the correctness of the strategies by fixing them separately. This induces two MDPs. If passing the strategy of the opponent to policy iteration for MDPs and after one iteration the algorithm terminates for both MDPs, we know that this must be the correct algorithm.

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:

PT

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]. We analyze the graph 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 compute all the strategies that correspond to the given bounds of the states. We can then fix each strategy separately, yielding two MDPs. We then perform one strategy iteration step on both MDPs. This step suggests to both MDPs the strategy of the opponent and fixes it, yielding two Discrete-Time Markov Chains (DTMCs). We then solve the DTMCs by non-iterative, exact methods as described in [BK08]. If the values of both DTMCs are equal, we have a guarantee that the strategies we would take are optimal, and we obtain the exact values of the states in the SCC.

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 solve MDPs, it is believed to perform worse than value iteration or strategy iteration and was not implemented in PRISM. We have implemented the approach to solve 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. Together with the topological option from [Kře+20], this yielded one of the most performant algorithms we present in this thesis.

4. Randomly generated 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 their structure can deviate from currently available case studies and thus express other structural features usually occurring. This can lead to a difference in algorithm performance and may get relevant if 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 can 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 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, or '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 timespan. 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 $\mathcal{SG}_{Connected}$.

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

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 assign states at random to either Maximizer or 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 $\mathrm{SG} \in \mathcal{SG}_{Connected}$.

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 is reaching 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.

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

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 $\mathsf{S}_{\square}$ 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 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 $m \in [M - |\mathsf{Av}(\mathsf{s})]$ * Add as many actions as possible * \
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

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 $\mathsf{S}_{\square}$ and $\mathsf{S}_{\bigcirc}$. Next, line 3 and 4 provide an initial state and a target. Thus, we only need to argue that Av is truly a mapping of $\mathsf{S} \to 2^{\mathsf{A}}$ 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 $\mathsf{S} \to 2^{\mathsf{A}}$.

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

To prove (i), note that whenever we introduce an action a to the set of enabled actions $\mathsf{Av}(\mathsf{s})$ of a state $\mathsf{s} \in \mathsf{S}$, we have $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}') = 0$ for all $\mathsf{s}' \in \mathsf{S}$. We increase $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}')$ 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, $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}')$ can only be increased 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)

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 $SG \in \mathcal{SG}_{Connected}$, we consider the following lemma:

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

Proof. **Show** $\mathcal{SG}_{Algo} \subseteq \mathcal{SG}_{Connected}$:

For this statement to hold, any $SG \in \mathcal{SG}_{Algo}$ 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 can reach 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 $\mathcal{SG}_{Connected} \subset \mathcal{SG}_{Algo}$:

Pick an arbitrary but fixed stochastic game $\mathrm{SG} \in \mathcal{SG}_{Connected}$. Next, we show that there is a run of our algorithm that will return a stochastic game $\mathrm{SG}' \in \mathcal{SG}_{Algo}$ where SG' is an automorphism to SG.

It is most likely wrong calling everything automorph. But it is also wrong referring to it by 'the same' (for example initial states). But I also do not want to make the proof unnecessary complicated. 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 $\mathsf{S}_\square$ and $\mathsf{S}_\circ$ are automorph for SG and SG'.
3. SG and SG' have automorph initial states and targets.
4. All state-action pairs in SG and SG' yield automorph probability distributions in δ.
5. Every state in SG and SG' have the automorph actions.

We decide randomly on the number of states in line 1 of Algorithm 1 and partition S into $\mathsf{S}_\square$ and $\mathsf{S}_\circ$ at random in line 2 of Algorithm 1. Thus, for every $\mathrm{SG} \in \mathcal{SG}_{Connected}$ there exists $\mathrm{SG}' \in$ *randomRandomOutcome* that have the same number of states to which there is an enumeration such that they are partitioned equally in both stochastic games. Both games have automorph initial states since the automorph mapping of the state indices can be used to determine the initial state of SG' according to SG. 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.

To show that statement 5 holds in addition, note that each $\mathrm{SG} \in \mathcal{SG}_{Connected}$ has a minimal set of state-action tuples such that the initial state is connected to every state. Taking this set, we can perform a breadth-first-search from the initial state to provide an enumeration of the states.

If we iterate over the states along 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 the initial, state 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 $(\mathsf{s}', \mathsf{a})$ can lead into arbitrary states and the probability distribution of $(\mathsf{s}', \mathsf{a})$ 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 $\mathcal{SG}_{Algo} = \mathcal{SG}_{Connected}$, in general Algorithm 1 does not sample $\mathcal{SG}_{Connected}$ 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 a 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. On one hand, this allows us to expose parameters to the user that can 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 on one hand 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 avoid too deterministic model generation.

Limits and additional guideline options

Although the parameters we expose for random generation can 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 and number 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 and number of SCCs in the model. Both guidelines are connected from the initial state, but are not able to create any game in $\mathcal{SG}_{Connected}$.

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, leafs are not required to have k actions. Their actions are only introduced during the backward process and are there to 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 use then 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 along 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 that is connected from the root where the user has an easy way of controlling the number and 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. 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 on how to use the modules and how to extend 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 a number of 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
- 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: How probable is it that a state generates actions during the backward procedure?
- branchingProb: How probable is it that an action has multiple transitions
- 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 precomputation. We introduce the switch to ensure that the generated models is not solved mostly during precomputation.

With correct implementation, generating models is slow. Say that you have a 'fast' switch to make randomPicker less random but way faster. With fast <1ŝ for a 10k model. Without its 10s

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

Here we describe what every module does.

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 in 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. I should probably rewrite this such that every guideline has its own parameter dataclass.

Lastly, there is a module called "randomStateGetter.py". This class implements various randomPickers. A randomPicker selects a state of the state-space randomly. This is used for example to decide which state should have an action to the currently introduced state. Different implementations allow 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 decided which state should receive an outgoing action, and another one that decides where a transition should lead. Should rename them to randomPicker and make them truly more random.

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. Should maybe put this in a table to make it easier to reference, find and read. The model features we track along with their names we use in the experimental section are:

State – related :

- (NumStates) Number of states
- (NumTargets) Number of targets*
- (NumSinks) Number of sinks*
- (NumUnknown) Number of unknown states*
- (NumMaxStates) Number of Maximizer states*
- (NumMaxStates) Number of Minimizer states*

Actions – related :

- (NumMaxActions) Number of maximum actions per state
- (NumProbActions) Number of actions with probabilistic 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

MEC – related :

- (NumMECs) Number of MECs

- (BiggestMEC, SmallestMEC) Size of biggest and smallest MEC*
- (AvgMEC, MedianMEC) Average MEC-size and Median MEC-size*

SCC – related :

- (NumSCCs) Number of SCCs
- (BiggestSCC, SmallestSCC) Size of biggest and smallest SCC*
- (AvgSCC, MedianSCC) Average SCC-size and Median SCC-size*
- (MaxSCCDepth) Longest chain of SCCs from the initial-state SCC
- (NumSCCNonSingleton) Number of SCCs with cardinality bigger than 1
- (SmallestSCCNonSingleton) Smallest SCC with cardinality bigger than 1
- (AvgSCCNonSingleton) Average SCC size with cardinality bigger than 1*

Path – related :

- (NearestTarget) Shortest Path from initial state to the nearest target
- (FurthestTarget) Shortest Path from initial state to the furthest target

The values of features with a star are measured absolutely, but displayed relative to a fitting measure. When referring to the relative value, we a % sign instead of the "Num"-prefix (e.g. Unknown% instead of NumUnknown). All relative measures that are related to states like number of unknown states are displayed relative to $|S|$ (NumStates). 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 **OVI**, $\mathbf{SI_{LP}}^{\mathbf{T}}$ 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 of 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, untrue 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

Include $\mathbf{SI_{VI}}$ as an okay alternative to BVI Try to verify that $\mathbf{SI_{LP}}$ may be bad. The best way to argue about this is that we get out-of-memory errors. This also is what Gandalf said about memory: LP takes more.

In this chapter, we analyze models regarding their feature distribution and algorithms regarding their performance. Furthermore, we investigate which model features influence the performance of the algorithms.

7.1. Experimental setup

Various algorithms we consider were already implemented in PRISM-games [Kwi+20]. We extended PRISM-games by the algorithms $\mathbf{SI_{LP}}$, $\mathbf{SI_{LP}}^{\mathbf{T}}$, and **PT**. Moreover, for **PT** we added precise Markov chain solving, which was not present in PRISM before, and extended strategy iteration (which was implemented in [Kře+20]) to use this precise solving. To solve Markov decision processes, we have used the academic license of Gurobi version 9[1]. 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 for all models. The memory limit for every experiment was 6 GB. For large model (ANY REF TO LARGE), we have set 36 GB memory limit and 30 minutes time limit for all models.

7.1.1. Case studies

We consider case studies from four different sources: (i) all real case studies that were already used in [Kře+20], which are mainly from the PRISM benchmark suite [KNP12]. For a detailed description of the real case studies see Appendix A. We omit models that are already solved by pre-computations. (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) To create models with at

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

least 1 million states, we prepend games as described in Section 4.4. The games we prepend are very similar to the RandomTree guideline from Section 5.1. (iv) randomly generated models generated by Algorithm 1 and our additional guidelines from Subsection 5.1.

Here we should state the exact parameters we have used for our models. States are 1000 to 10000, transition probabilities between 0.1 and 0.01 etc. However, since there are many parameters and many models I probably rather put it into the appendix

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, we could not generate models with larger state spaces (at least 1000000 states) because every action in our random generated models is stored explicitly in the corresponding .prism file. Thus, PRISM requires a lot of time to parse these files. Parsing the randomly generated models with 10000 states takes up to 200 seconds and scales up non-linearly with the number of actions or states in our models.

Handcrafted models and real case studies achieve a larger state space because many the .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. However, precomputation for these models requires more than 1 hours of computation time, so we include them without precomputation. Thus, our benchmarking set for large models is created only out of games we generate at runtime and the real case studies that are non-trivial and parameterizable, which are AV, dice, and hallway. Additionally, we may add in MulMEC and BigMEC without precomp, since for them precomp also takes too long.

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. 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 lines extended by the whiskers mark the 10 and 90 percentile. Dots outside of the whiskers represent outliers that differ significantly from the rest of the dataset. The are plots 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

This is very abstract, and I do not like it. If you use it only once you can move it down OR put it into an own subsection OR reference it only once In some cases we want to analyze how two sets of events **A**, **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. Next, we study the properties of the models in **A** and the properties of the models in **B** by using a 1-dimensional scatter plot. When using this plot, we try to identify whether models in **A** distribute differently along the spectrum of feature values than **B**. For example, we may find that all models in set **A** have smaller MECs than models in **B**.

7.2. Model analysis results

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 plot provides the following insights:

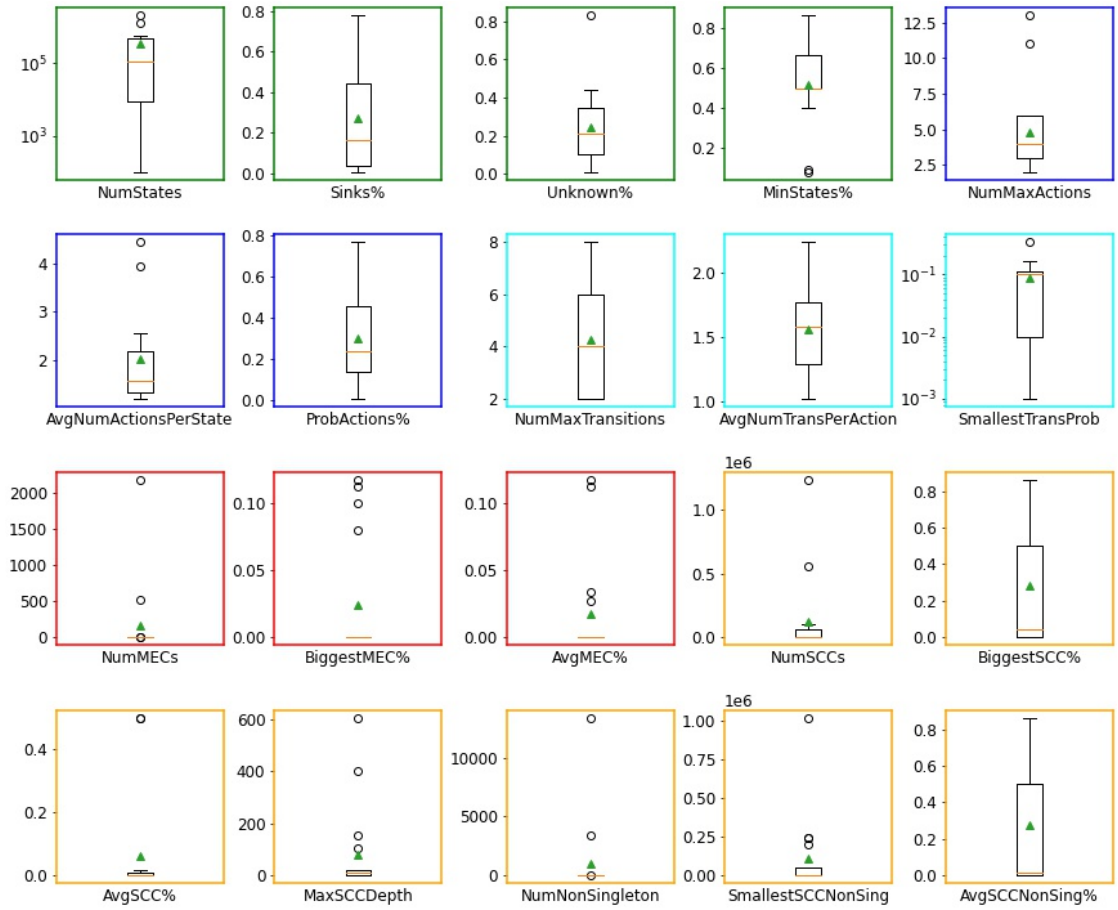


Figure 7.1.: A Box plot of the feature distribution of the real case studies. The description on how to read the plot is provided in Section 7.1.2.

- According to AvgNumActionsPerState and AvgNumTransPerAction, models have on average 2 actions per state and 1.5 transitions per action
- According to NumUnknown, on average 80% of the states of the models are trivial, and their value is computed with simple graph algorithms
- Generally, the number of states is evenly split between Maximizer and Minimizer

- According to NumProbActions, usually around 70 to 85% of all actions are deterministic
- According to NumMECs, most models do not contain end components

By furthermore printing 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: Leave out? No real gain here

- Models with numerous actions per state
- Models with numerous Transitions per action
- Models with very small transition probabilities

Furthermore, we use box plots to evaluate for which features our random generation algorithm from Chapter 4 is biased. Figure 7.2 contains the box plots for our randomly generated models.

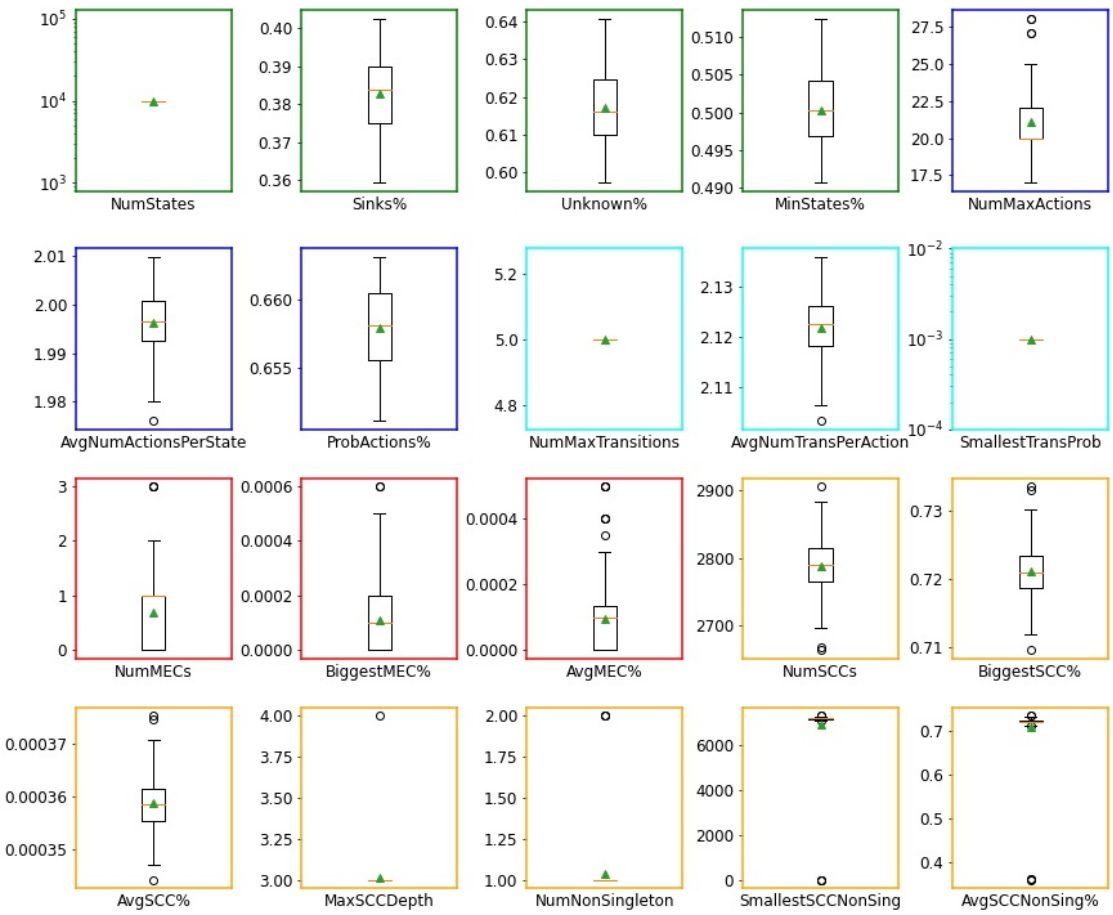


Figure 7.2.: A Box plot of the feature Distribution of models generated randomly with Algorithm 1. The description on how to read the plot is provided in Section 7.1.2.

The biases we read from this plot are:

- On average, 38.5% of the values of states of the models are computed by trivial graph algorithms. NumSinks 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. However, our parameters allows 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 such that we form bigger MECs. For example setting the parameters in such a way that the actions are deterministic and adding an action to every state in the backwards procedure of Algorithm 1 creates models with a high tendency of forming few MECs that usually contain the whole state-space. Nevertheless, without providing a specific guideline, we have very limited control over the number and size of the MECs.

- 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. 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 way in which 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. We use the same parameters to create the models as for the set displayed in Figure 7.2, but change the number of actions to Parameter. Figure 7.2 contains the box plots for our randomly generated models.

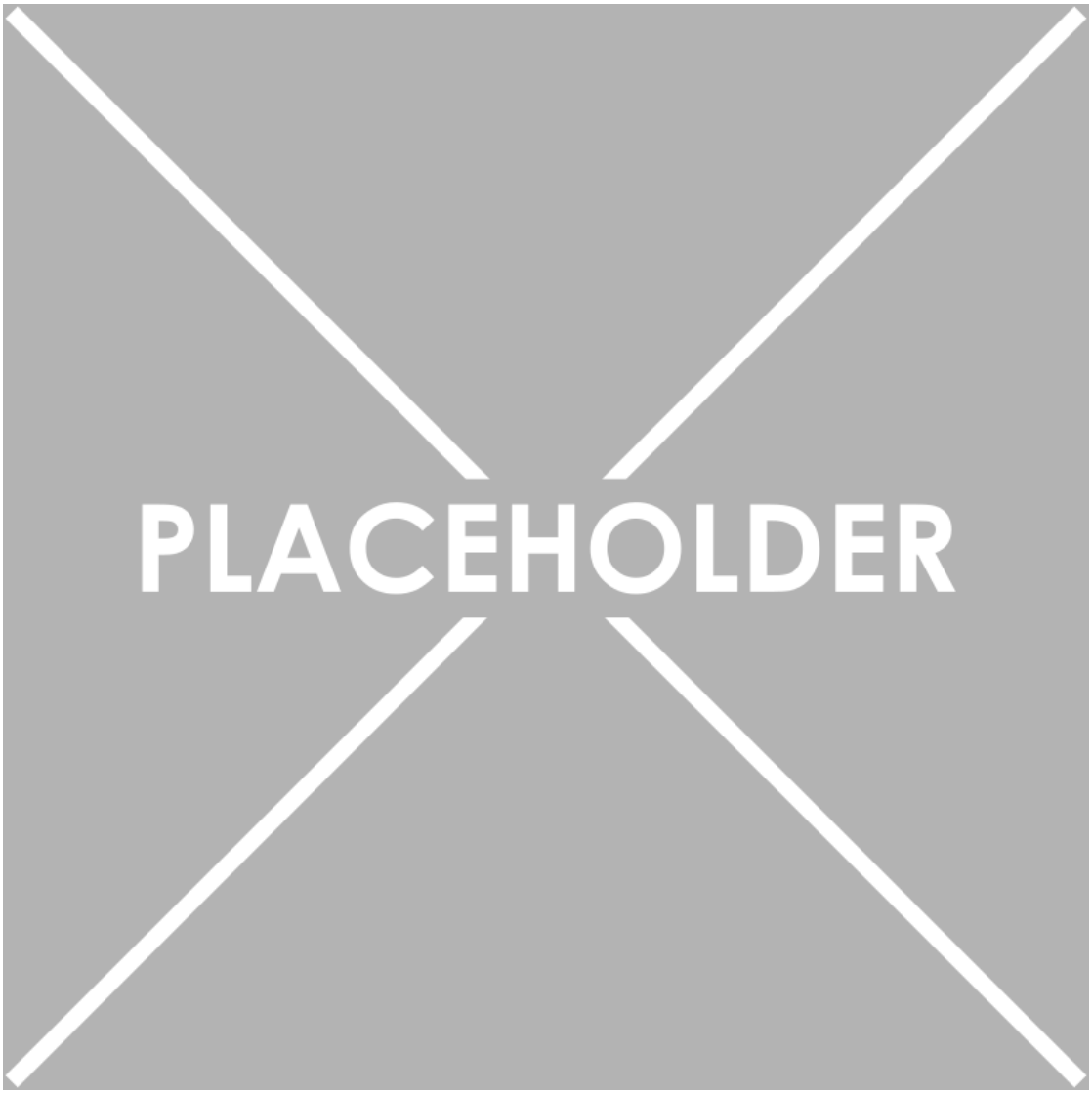


Figure 7.3.: Feature Distribution of randomly generated models by applying the RandomSCC guideline. For a description on how to read the plot is provided in Section 7.1.2. The evaluation of the plot is located in Section 7.2.

Some easy conclusions. Cleary different - smaller SCCs, more actions, hopefully less bias towards states with smaller index

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 over 60% of the states could not be computed during precomputation, for the guidelines RandomSCC and RandomTree on average less than 10% of the state space could not be 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 through RandomSCC and RandomTree harder, we set the forceUnknown switch to true. At the moment this is equivalently to disabling the computation of states that can surely reach the target. So then the question is why didnt we just do that? Because setting the value to 0.9 is a realistic scenario. There can be models that have value 1 only in the target. But this feels like a stupid way of justifying this switch. In the end the states are still kinda trivial, right? The low Unknown% is mostly due to large parts of the state space being unable to ever reach a target.

7.3. Algorithm comparison results

In this section, we compare the Algorithms introduced in Section 2.7 on real case studies, handcrafted 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.

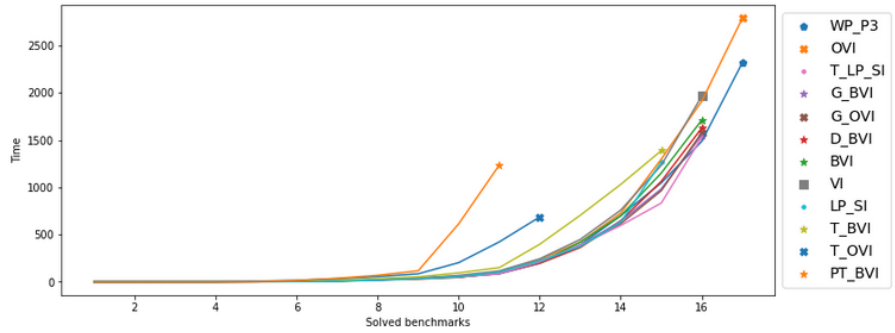
First, we provide a general overview of the performance of all algorithms on our benchmarking set through a line plot in Figure 7.4.

split into real case studies (Figure 7.3a) and randomly generated models (Figure 7.3b)We read several clues from Figure 7.4: **WP**, **OVI**, and $\mathbf{SI_{LP}}^{\mathbf{T}}$ seem to be the most performant algorithms for our benchmarks.

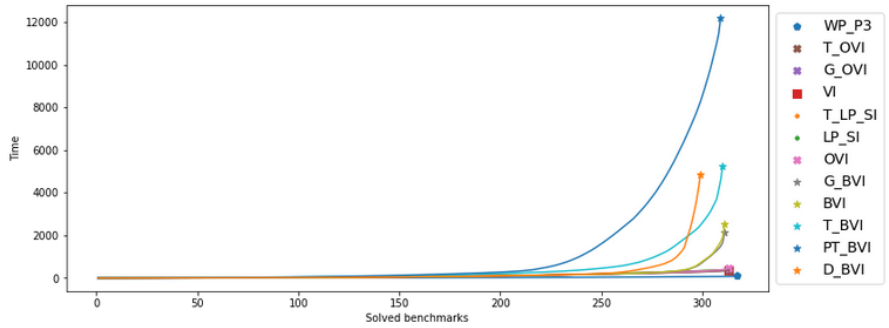
We split our analysis into three subtopics: First, we compare the three value-iteration-based algorithms with guarantees **OVI**, **WP**, and **BVI**. Then, we analyze the impact of the optimizations from Subsection 2.7.4 on their respective baseline algorithms. Lastly, we investigate the performance 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 optimized the implementation of the algorithm is.

7.3.1. BVI vs OVI vs WP

We read from Figure 7.4 that WP is the fastest to solve the problems on both random-generated models and real case studies. To see whether this is the case for all models or only when accumulating runtime, we consider the scatter plot in Figure 7.5. The x-axis marks the time **WP**



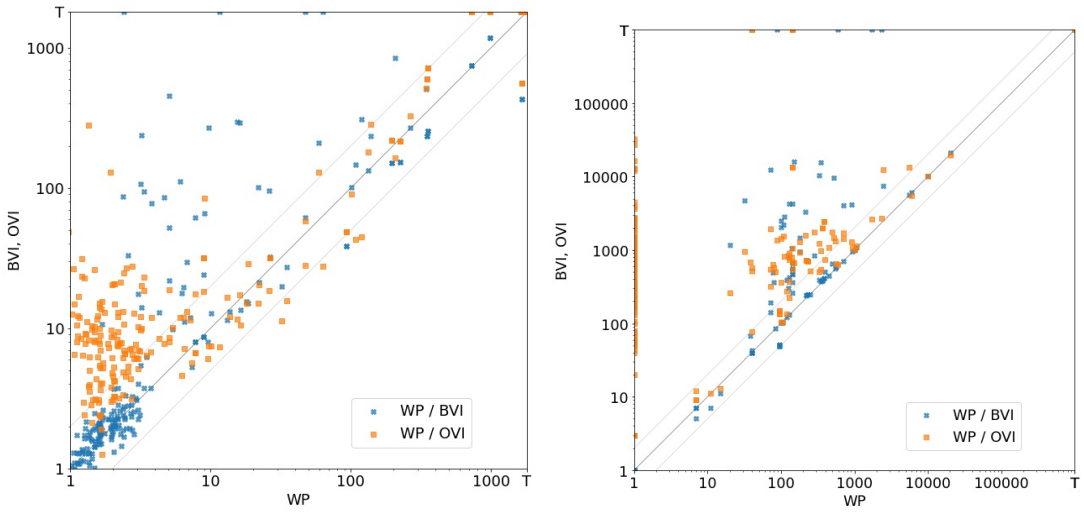
(a) Accumulated algorithm performance overview on real case studies



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

Figure 7.4.: A line plot providing an overview of the accumulated algorithm performance.

requires to solve a stochastic game, and the y-axis marks the respective time **BVI** or **OVI** needs. The two lines next to the diagonal mark the case that **WP** is twice as fast as **BVI / OVI**s 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**. See 7.1.2 for a reference on how to read the plot.

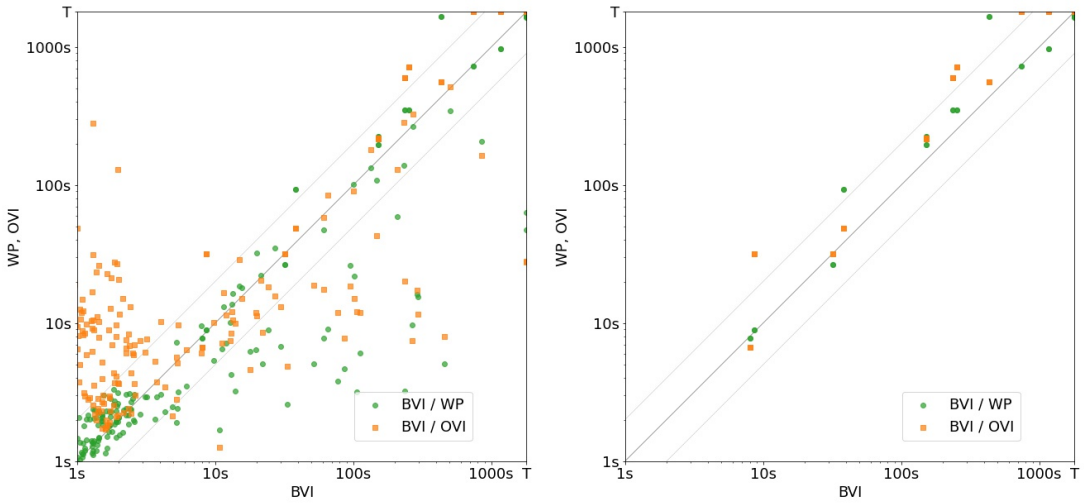
WP

Regarding time, **WP** is neither significantly better nor worse than **OVI** and **BVI**. However, in every data set we analyzed, **WP** was one of the fast algorithms when measuring accumulated time. On large models, **WP** was accumulated slower than **BVI**, but still comparable. When comparing iterations, **WP** requires almost always fewer iterations than **BVI** and **OVI**, and never more than twice as many.

Since we could not find a weakness of **WP**, we conclude that it is a good 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 the left scatter plot with all the models of Figure 7.6 suggest that for more complicated models **OVI** and **WP** are faster than **BVI**, the times required to solve large models only in 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 large models.



(a) Time required to solve a model with all models

(b) Time required to solve a model only with large models

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. I do have an image of that, but it is probably not interesting to see. Appendix or leave out?

OVI

While **OVI** seems very performant when looking at accumulated runtime, Figure 7.6 and Ref Large models provide evidence that it requires more time than **BVI** and **OVI** for large models. This is due to the termination criterion of **OVI**: If no value improves more than an ε', the verification phase starts to ensure that **OVI** is close enough to the optimal value. If the verification fails, ε' is halved. Thus, it is possible that the verification phase fails, although the game is almost solved. In that case **OVI** will iterate unnecessary extra steps until it reaches its new precision that verifies that **OVI** may indeed terminate. For large models, it is likelier that a model is so complex that it requires multiple verification phases as Figure Show something about verification phases (or do not) suggests. Thus, it is also more probable that the precision is halved at a point that will lead to unnecessary iterations. In addition to that iterations are also more costly for larger models since the state space is bigger.

OVI requires generally require more iterations than **WP** and **BVI**. This is partly because of the extra iterations that may happen in **OVI**, and partly because during verification phase the iteration counter increases, but only the lower bound is being iterated. 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 7.7 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 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).

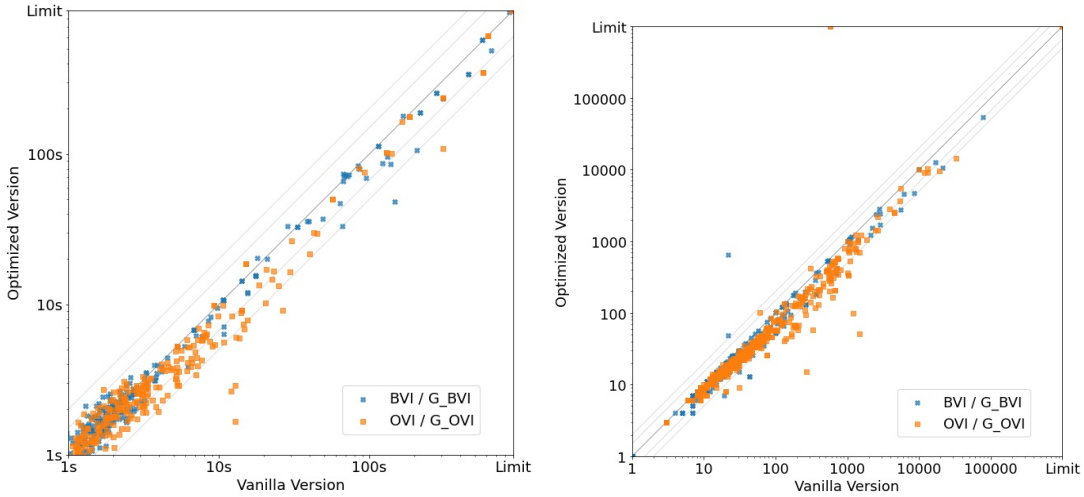
Clearly, in some instances where a game has value 0 or 1, **WP** is able to identify its value in only 1 iteration while **OVI** cannot. **BVI** is also capable of solving these models in 1 iteration. Ref to BVI scatter In the runtime scatter plot of Figure 7.5, these models are the cloud of orange dots in the lower left quadrant Is that true?. However, even if **BVI** and **WP** require only one iteration, they still require up to 5 seconds to perform this iteration.

I could talk about best-case OVI examples one action leading to target with 0.5 in every state and the other into the chain with self loop.

7.3.2. Analyzing the optimizations

Figure 7.4 indicated 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. To each optimization, we provide a scatter plot with 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**:



(a) Time required to solve a model

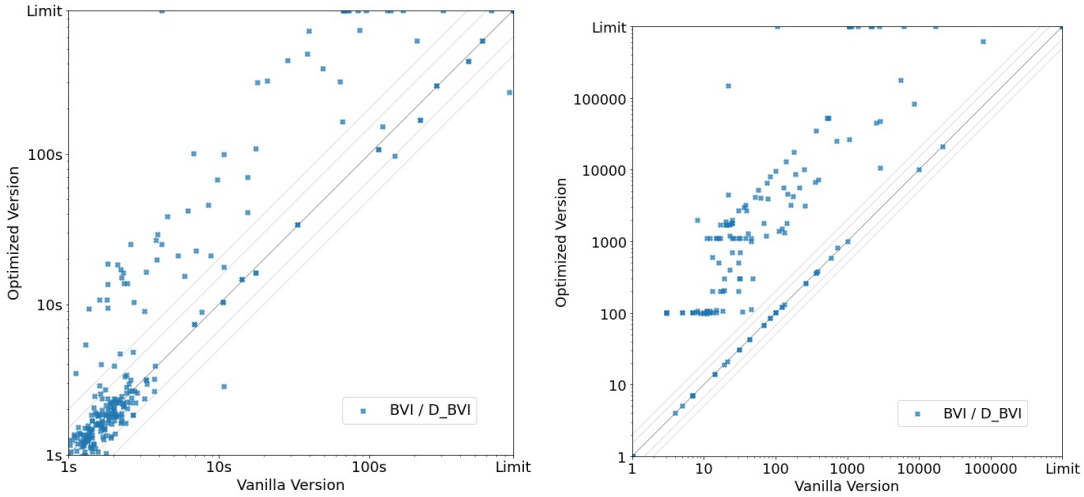
(b) Iterations required to solve a model

Figure 7.8.: **BVI** and **OVI** compared to their Gauss-Seidel optimizations

CAREFUL! For big models this may be untrue As Figure 7.8 suggests, using the Gauss-Seidel optimization may reduce both the time and iterations required to solve a model by 1-4 times in most cases. However, Figure 7.4 shows when accumulating runtime, **OVI** is still slightly faster than $\mathbf{OVI_G}$. 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 turn out to be faster sometimes. Furthermore, it is possible that there are more iterations required to solve. 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 less iterations. Changing the order of computation of the states for the Gauss-Seidel optimization by computing along a topological enumeration of the states did not yield improvements in our experiments.

Probably, Gauss-Seidel should be even worse for bigger models since there is more room for "not paying off" using state-by-state computation.

D for **BVI**:



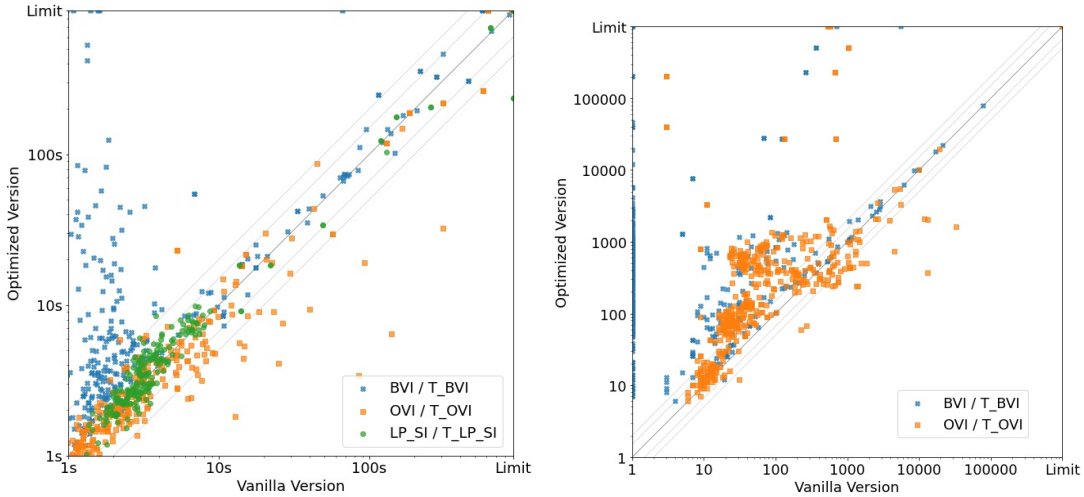
(a) Time required to solve a model

(b) Iterations required to solve a model

Figure 7.9.: **BVI** compared to its optimization where deflating happens only every 100 iterations

Figure 7.9 clearly indicates that although $\mathbf{BVI_D}$ may solve sometimes models faster, for most of our models it could not compete with **BVI**.

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



(a) Time required to solve a model

(b) Iterations required to solve a model

Figure 7.10.: **BVI**, **OVI** and $\mathbf{SI_{LP}}$ compared to their topological optimizations

As the scatter plot 7.10 shows, the topological addition for strategy iteration with linear programming in the real case studies and random models does neither in- nor decrease the performance of the algorithm considerably. However, most models have very few SCCs, so the topological optimization does not contribute a lot. The data point where $\mathbf{SI_{LP}}^{\mathbf{T}}$ is significantly faster is on the real case study "dice", where every state is an SCC on its own. This is obviously the best-case scenario for topological algorithms. Show Scatter. Explain why **OVI** and **BVI** do not scale that well (explained in Gandalf Paper)

When comparing **OVI** with $\mathbf{OVI}_T$ I could not find any defining feature that would say much **PT** for **BVI**:

PT seems to perform worse than **BVI** in general. This is because we solve the DTMCs with exact methods. Scatter BVI vs TOP. This requires a matrix inversion, which is an $\mathcal{O}(n^2)$-operation, where n is the number of states in the SCC whose value is being computed. The bottleneck becomes apparent if we consider a scatter plot where we show the runtime against the size of the biggest SCC as in Figure **??** ADD FEATURE-SCATTER. We have also tried solving the resulting MDP with linear programming instead of strategy iteration, but it was still worse than $\mathbf{SI_{LP}}^{\mathbf{T}}$.

7.3.3. Algorithms based on strategy iteration

It would also be interesting to see if more probabilistic loops affect $\mathbf{SI_{LP}}$ as strong as VI.

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

Strategy iteration with linear programming

$\mathbf{SI_{LP}}^{\mathbf{T}}$ 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 $\mathbf{SI_{LP}}^{\mathbf{T}}$ this is linear programming.

Linear programming scales worse than value iteration for huge models. As Figure REF BIG MODELS indicates, $\mathbf{SI_{LP}}^{\mathbf{T}}$ and $\mathbf{SI_{LP}}$ were slower to solve the models than **OVI**, **BVI** and **WP**. This is partly due to the LP solver running out of memory, which happened 8/36. Check that this number is correct.

To test this, we have run several benchmarks on models with state size 10 million and varying SCC sizes. [Now enter Feature-Performance Scatter Plot] The bigger the SCCs become, the slower $\mathbf{SI_{LP}}^{\mathbf{T}}$ becomes. At a size of [...] per SCC, **WP** was faster than $\mathbf{SI_{LP}}^{\mathbf{T}}$. Thus, we do not recommend using $\mathbf{SI_{LP}}^{\mathbf{T}}$ 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. Also, the topological improvement allows solving models with huge numbers of small SCCs faster than value iteration. And also value iteration was the focus of research for the last 20 years. LP could likely be improved. At the moment, we do not even deflate but use MIP to encode the maximum-best-exit constraints. But this should maybe go into future work

Strategy iteration with exact Markov chain solving

7.3.4. Big models

Might also put in somewhere else. Basically too little data but also pretty though PRISM constraints. $\mathbf{SI_{LP}}^{\mathbf{T}}$ ran sometimes out of stack (see memo comparation in GANDALF for QP or make own) suggesting that $\mathbf{SI_{LP}}$ may not scale too well for bigger models but it may be good if there are many SCCs with smaller SCCs. It seems that **OVI** does not scale too well. However, in the end it is very important to note that we are lacking the capacity to create big models.

8. Conclusion and future work

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

Our randomly generated models can be in adjusted in a way that resembles the currently available case studies we are aware of. At the same time, we are able to adjust the parameters and guidelines in such a way that they do express structural properties that differ from the case studies. The only limit we face at the moment is that explicit models can be parsed only slowly in PRISM. To avoid this issue, we provide the option to prepend graphs at runtime.

Furthermore, we explored were helpful to analyze the growing set of both models and algorithms to analyze. Establishing analysis tools allowed us to prototype algorithm ideas, assess fast whether it seems promising, and provide a better overview of the data we collect. Additionally, our tools for model analysis improve with increasing data points. This is not the case for tables as visualization and analysis tool, but they are nevertheless frequently the only utility used for visualization in analysis [Kel+18][Pha+20][Brá+14].

Lastly, we have compared algorithms for SG solving and have found that although $\mathbf{SI_{LP}}^{\mathbf{T}}$ is not competitive to **OVI**, **BVI** and **WP** for large models, it may be a good alternative for games that are adversarial to approached based on value iteration. Furthermore, it may be a good approach in case a model has many SCCs that are all of small size.

Like $\mathbf{SI_{LP}}^{\mathbf{T}}$, **OVI** seemed promising for smaller models, but is likely not competitive with **BVI** or **WP** for most large models. Thus, for now we conclude in general one should try **BVI** or **WP** when trying to solve a stochastic reachability game, and should pivot to other methods if these approaches struggle to converge.

Future work

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 when to use **WP** instead of **BVI**.

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 more research of 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 adapt algorithm to solve the problem. With enough models, the decision which algorithm to use may be made by a neural network. Also, combinations of algorithms like **OVI** and **BVI** could then 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 to our random generation algorithm that would hold chunks of the data in implicit blocks to make the files easier parsable.

Lastly, 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. I may also leave this out since I have nothing to say about it. We would simply generate less junk and more of the state space would be used in a smart way instead of wasting it on trivially computed states we could have summarized into sinks and targets all along.

List of Figures

Bibliography

[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. 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.

[CC15] K. Chatterjee and M. Chmelík. "POMDPs under Probabilistic Semantics." en. In: *Artificial Intelligence* 221 (Apr. 2015), pp. 46–72. ISSN: 00043702. DOI: 10.1016/j.artint.2014.12.009.

[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+16] K. Chatterjee, M. Chmelik, R. Gupta, and A. Kanodia. "Optimal cost almost-sure reachability in POMDPs." In: *Artif. Intell.* 234 (2016), pp. 26–48.

[Cha+ed] K. Chatterjee, J.-P. Katoen, M. Weininger, and T. Winkler. "Stochastic Games with Lexicographic Reachability-Safety Objectives." In: *CAV 2020*. Lecture Notes in Computer Science. Springer, accepted.

[Che+11] 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+13a] 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.

[Che+13b] T. Chen, M. Z. Kwiatkowska, A. Simaitis, and C. Wiltsche. "Synthesis for Multi-objective Stochastic Games: An Application to Autonomous Urban Driving." In: *QEST*. 2013, pp. 322–337.

[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.

[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." In: *Electronic Proceedings in Theoretical Computer Science* 326 (Sept. 2020), pp. 131–148. ISSN: 2075-2180. DOI: 10.4204/eptcs.326.9.

[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.

[KY76] D. Knuth and A. Yao. "Algorithms and Complexity: New Directions and Recent Results." In: Academic Press, 1976. Chap. The complexity of nonuniform random number generation.

[LCK95] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. "Learning Policies for Partially Observable Environments: Scaling Up." In: *ICML*. Morgan Kaufmann, 1995, pp. 362–370.

[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: Additional benchmarks

ADD THE OTHER MODELS ASWELL We provide benchmarks similar to those in Section **??**. We compare Condon's quadratic program, value iteration, and our improved quadratic program, but we use other models. Each of these models is a stopping game.

Models

We use the models Dice [KY76] and Charlton[Che+13b], which are included in PRISM-games [Che+13a]. Furthermore, we use the models Hallway and Avoid the Observer, provided in [Cha+ed]. The descriptions of the models are also from [Cha+ed].

Dice [KY76]: First player 1 throws a fair die repeatedly until accepting an outcome. Up to N tries are possible. Then player 2 is allowed to throw the die at most as many times as player 1 did. A player wins if it achieves a higher outcome than the opponent; draws are possible. The number of throws N in this game is a configurable parameter. We compute the probability that player 1 wins and the probability that player 2 wins.

Charlton [Che+13b]: This model describes an autonomous car navigating through a road network. We compute the maximal probability of reaching the destination.

Hallway (HW) [Cha+ed]: This instance is based on the Hallway example standard in the AI literature [LCK95; Cha+16]. A robot can move north, east, south or west in a known environment, but each move only succeeds with a certain probability and otherwise rotates or moves the robot in an undesired direction. The example was extended by a target wandering around based on a mixture of probabilistic and demonic non-deterministic behavior, thereby obtaining a stochastic game modeling for instance a panicking human in a building on fire. Moreover, in assume a 0.01 probability of damaging the robot when executing certain movements; the damaged robot's actions succeed with even smaller probability. The primary objective is to save the human and the secondary objective is to avoid damaging the robot. We use square grid-worlds of sizes 8×8 and 10×10 and only compute the probability that the robot saves the human, since this is the only reachability objective.

Avoid the Observer (AV) [Cha+ed]: This case study is inspired by a similar example in [CC15]. It models a game between an intruder and an observer in a grid-world. The grid can have

different sizes as in Hallway. The most important objective of the intruder is to avoid the observer, its secondary objective is to exit the grid. We assume that the observer can only detect the intruder within a certain distance and otherwise makes random moves. At every position, the intruder moreover has the option to stay and search to find a precious item. In our example, this occurs with probability 0.1 and is assumed to be the third objective. We compute with the quadratic program only the maximal probability that the intruder exits the grid and that he finds a precious item on a 2×2 grid, since the first property is not a reachability, but a safety game.