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

Abstract

Simple stochastic games are a formalism used in modeling stochastic systems. We consider reachability games, where the goal of the first player is to reach a set of targets. 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 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 well with growing numbers of both the number of stochastic games we solve aswell as the number of algorithms we analyse. Lastly, we draw conclusion based on the now facilitated analysis about the performance of some of the prominent algorithms.

Contents

1. Introduction

We as a society 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 system and the program is doing exactly what it should and we expect it to.

Verification is a field of computer science dealing with this question. One widely used approach to do so is *model checking* where a certain real system is simplified to a theoretical model. The model is then checked for correctness of certain behavior and the result can then be applied to the real world.

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 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 (or policy) iteration and value iteration. In practive value iteration is considered the fastest of the three solution techniques. However, there is an adversary example [HM18] proving that value iteration may need exponentially many steps to solve a problem.

To all of the three techniques for solving reachability games there are many optimizations and changes available that affect their performance. However, at the moment there is very few data that allows for a solid quantative assertion of the performance of the solution techniques and their extensions. In fact, there are only around 12 real-world case studies for stochastic reachability games in total. Since every algorithms performance depends on the underlying structural properties of the stochastic game at hand, an algorithm that performs well on the available case studies could still overall be a bad choice since the the dataset might contain not enough structural varience to enable an adequate evaluation of the algorithm performance.

The contributions in this thesis are the following:

- 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
- An analysis of whether the real case-studies are biased towards certain graph structures
- A comparison of currently prominent solution techniques for stochastic games
- An evaluation of how several structural properties in stochastic games affect the performance of available solution techniques

Note that we will only focus on value iteration, strategy iteration and their extensions since it was already shown in [**Gandalf**] that at the moment Quadratic Programming is in general not a recommended approach to tackle reachability problems in stochastic games.

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. While Chapter 4 provides the benefits aswell as the theoretical aspects to random generation of stochastic games, Chapter 5 contains implementation details and a manual on how to use our implementation. In Chapter 6 we head into the analysis of stochastic games and solution technique performance. Chapter 7 presents benchmarks of the algorithms on the reachability games, aswell as results of the model and algorithm analysis. Lastly, Chapter 8 draws conclusions about our work and the resutls. Also we consider future work in this are of the thesis.

Related work

SGs are a generalization of Markov Decision Processes, which were the first to be introduced. Markov Decision Processes are a generalization of Markov Chains [Put14]

[GS06, Ch. 11].

The first to introduce the concept of stochastic games aswell as value iteration as solution algorithm was Shapley in [Sha53]. Condon has shown that solving simple stochastic games has a complexity in **NP** ∩ co-**NP** [Con92]. Condon has also introduces 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][**https://pubsonline.informs.org/doi/abs/10.1287/mnsc.12.5.359 - Strategy Iteration for MDPs**]. 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

[**GANDALF**] 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.

There are various model checkers for probabilistic model verification. Could mention others here like STORM We use PRISM-games [Che+13a] for model checking SGs and for the tracking of model and algorithm features. PRISM-games is an extension of the PRISM [KNP11] model checking tool for games. The random models we procude are stored in the typical PRISM file-format for models.

2. Preliminaries

- Explain what an "incoming transition" is
- What does it mean if a state can be reached? <-> it has incoming transition

In this chapter we introduce the underlying definitions necessary to have an understanding of the notation and foundation 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 cases 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 2.6.

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 $\mathsf{S}_\circ$. 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 $\mathsf{s} \in \mathsf{S}_\circ$ 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 perfectly?*

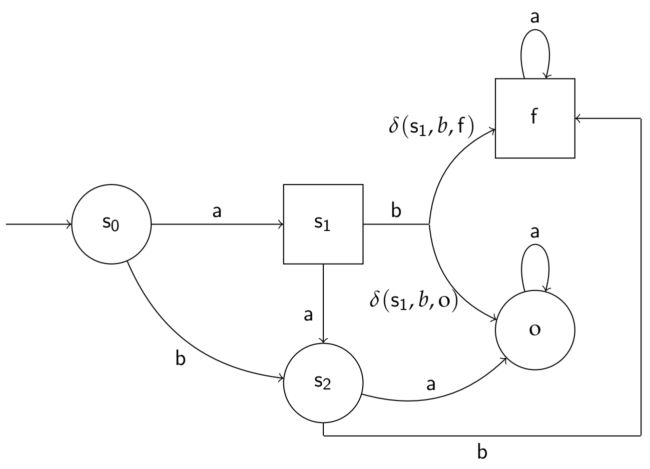


Figure 2.1.: An example of a simple stochastic game. $s_0, s_1, s_2, \mathsf{f}, \mathsf{o}$ are states. a, b are actions. s_1 and f are Maximizer-states, while the rest of the states belongs to the Minimizer. f is a target and o is a sink. $\delta(\mathsf{s}_1, b, \mathsf{f})$ and $\delta(\mathsf{s}_1, b, \mathsf{o})$ are the transition probabilities that a token in s_1 moved through action b would either be moved to f or o

2.2. Stochastic games

We define now simple stochastic games, also referred to as stochastic games [Sha53]. To do so, we need to introduce *probability distributions*. A probability distribution on a finite set X is a mapping $\delta : X \to [0, 1]$, such that $\sum_{x \in X} \delta(x) = 1$. The set of all probability distributions on X is denoted by $\mathcal{D}(X)$.

Definition 2.2.1 (SG). *A* stochastic game (SG) *is a tuple* $(\mathsf{S}, \mathsf{S}_\Box, \mathsf{S}_\circ, \mathsf{s}_0, \mathsf{A}, \mathsf{Av}, \delta)$ *where* S *is a finite set of* states *partitioned*[1] *into the sets* $\mathsf{S}_\Box$ *and* $\mathsf{S}_\circ$ *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* $\mathsf{S}_\circ = \emptyset$*, 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], which is equivalent to Condon's model of SGs.

The set of successors that can be reached from a state s taking the action $\mathsf{a} \in \mathsf{Av}(\mathsf{s})$ is

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

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 $\mathsf{Av}(\mathsf{s}) \neq \emptyset$.

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* $\mathsf{o} \in O \subseteq \mathsf{S}$. Thus, once the token reaches any sink o 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 o 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 o.

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].

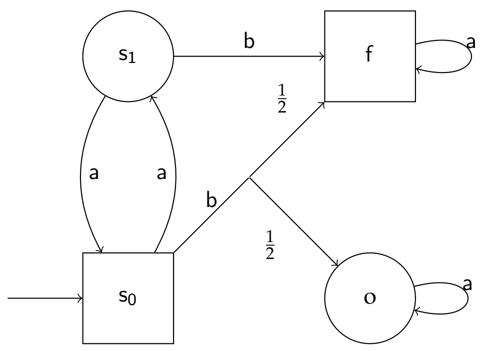


Figure 2.2.: An example of a non-stopping SG with an end component of size 2. If in s_0 action a and in s_1 action a are chosen for some strategies σ, τ the game does never stop. $\mathsf{V}_{\sigma,\tau}(s_0) = 0$ because the probability to reach any $f \in F$ is 0 with these strategies.

However, we do not need this level of detail for this thesis and will avoid it for easier readability. We use $\mathsf{V}(\mathsf{s}, \mathsf{a})$ as the value of s for a strategy in which s chooses a.

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 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 [Sha53] (see also [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 an 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 0 because the best that the Minimizer can do is to take action b which will always lead to s_2 and then get with action a to o. For this strategy τ, the Maximizer can not win.

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 o.

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 $\mathsf{w} = \mathsf{s}\mathsf{a}_0 \ldots \mathsf{a}_n\mathsf{s}' \in (T \times B)^* \times T$, i.e. the path stays inside T and only uses actions in B.*

Figure 2.2 illustrates a simple SG with an end component. 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 an 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 of at least size 2 is for many algorithms a non-trivial task that requires additional concepts.

2.6. Algorithms for Stochastic Games

Next, we consider 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 solve stochastic games are *Value Iteration*, *Strategy (or Policy) Iteration* and *Quadratic Programming*.

2.6.1. 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 O and finally the remaining states $\mathsf{S}^?$. For T and O (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. [**visurvey**]:

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

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

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 [**visurvey**] (formally $\lim_{i \to \infty} \mathcal{B}^i(\mathsf{L}_0) = \mathsf{V}$).

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 [**bvi**; **atva**; **KKKW18**]) 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 to some greater fixpoint instead due to end compoments. See [**KKKW18**] for an example. To fix this issue, **BVI** searches for the end components where all states agree on the same exit out of the end component. The upper bound for these states can then be defined by the upper bound of the states outside of the end component. This approach is usally referred to as *deflating* [Kel+18]. With deflating, the upper bound decreases monotonically towards the least fixpoint. Once $\forall \mathsf{s} \in \mathsf{S} : \mathsf{U}(s) - \mathsf{L}(s) \leq \epsilon$, **BVI** terminates and reports $\mathsf{s}_0) = \frac{\mathsf{U}_0 + \mathsf{L}_0}{2}$.

Optimistic Value Iteration **OVI** takes a similar approach, but instead of calculating the upper bound at all times along the lower bound, a bound B is guessed from the lower bound by adding a small ϵ to the lower bound. the upper bound is guessed from the lower bound by adding a small ϵ to the lower bound. If this induced bound B can

[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.

be verified to be an upper bound, i.e. $\forall s \in S : \mathcal{B}(B)(s) \leq B(s)$, 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.

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 counter examples where value iteration requires exponentially many steps to converge and its performance is worse than the other solution approaches like strategy iteration.

There are more extensions and optimization to value iteration like SVI? Learning based approach?. The optimizations we consider in this thesis are:

- Talk about TOP, WP and T. No need to talk about G and D since these are rather implementations.

2.6.2. Strategy Iteration

In Strategy Iteration, each player starts out with an arbitrary strategy. In turns, the Maximizer and the Minimizer adjust their strategies to improve their winning probability [Con93].

Whenever a player fixed a strategy, the resulting SG is simplified to an MDP. From there on, any solution method for MDPs can be applied. We solve the MDP with linear programming **[Citation to any LP-solving explanation for MDPs. Maybe Puterman?]**

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 there is no example that can confirm that there is a stochastic game where strategy iteration truly needs exponentially many iterations.

2.6.3. Quadratic Programming

In Quadratic Progamming, 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, **[Gandalf]** 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.

Algorithm Performance and Stochastic Game Structure

We believe that 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. As has been shown in [**GANDALFpaper**] and [HM18], there are certainly some stochastic games that can be easy or very hard for certain algorithms. However, at the moment there are only few insights on which structural properties should be taken into account when deciding on which algorithm to take, and which should not.

Thus, we analyse 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:

TOP

TOP as in *topological and precise* is an extension to Value-Iteration-based algortihms that builds on top of the idea of topologicals sorting of [**Gandalf**]. We analyse the graph underlying the stochastic game, detect the strongly connected components (SCCs), 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 both a guarantee that the strategies we would take are correct aswell aswell as the exact values of the states in the SCC.

Widest Path for Prism3

Widest Path was introduced as an alternative to solving maximal end components for bounded value iteration Make sure you have introduced BVI[**WidestPath**]. We reimplemented it in the version of PRISM.

Linear Programming for MDPs

We have implemented the approach to solve MDPs with linear program as in [**ANY BOOK**]. This allows for a combination of strategy iteration to fix one players strategy, and linear programming to solve the induced MDP. Together with the topological option from [**GANDALF**], this yielded one of the most performant algorithms we present in this thesis.

4. Randomly Generated Models

4.1. Why Use Randomly Generated Models

Using randomly generated problems is a classic approach to test algorithms. A key benefit of randomly generated models is that by not being to a real problem, their structure can deviate from currently available case studies and thus express other structural features. This can lead to difference in algorihtm performance and may get relevant if new case studies emerge. Also, randomly generated models are 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 have often parameters that can be adjusted, resulting in infinetely 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.

Clearly, 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 (like the haddad-monmege model does for value iteration), but contributes only very little to the overall coverage of possible reachability-problems for stochastic games.

To tackle this issue we introduce randomly generated models.

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 model in which the inital state is connected to 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 could by simplified by introducing a single sink, and thus add unnecessary complexity. We refer to the set of stochastic games where the inital state is connected to every other state in the game by $\mathcal{SG}$.

Ideally, we would like sample uniformly from $\mathcal{SG}$ to minimize the number of structural biases in our models. However, since it was not clear for us how to achieve this, we use instead an iterative generation approach. While not samling $\mathcal{SG}$ space uniformly, it is easy to implement and modify. Also, since $\mathcal{SG}$ is infinite, as far as I understand there is no way to sample the space at uniform. This is at least what this random 'Mathematician' in this post says https://www.askamathematician.com/2010/01/q-is-it-possible-to-choose-an-item-from-an-infinite-set-of-items-such-that-each-one-has-an-equal-chance-of-being-selected/ But it makes sense to me.

4.3. Our take on creating models as random as possible

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?

We use Algorithm **??** to create any random stochastic game that is connected from the inital state. In the forward procedure, we iterate over every state $s \in S$ 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 is connected to every state in the stochastic game. The backward procedure then adds arbitrary actions to arbitrary states to enable generating every possible $SG \in \mathcal{SG}$.

To generate the actions of a state, we use Algorithm **??**. It receives a state-action pair to which the underlying transition probability distribution sums up to less then 1. It then adds transitions to not yet reached states by increasing their transition probabilities at random. This is repeated until the state-action pair has a valid transition probability distribution.

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

Proof. S is finite since its size is determined by a random number $n \in \mathbb{N}$. Thus, we only need to argue that Av is truly a mapping of $S \to 2^A$ and that the transition function yields a probability distribution. Whenever we introduce state-action pairs,

Algorithm 1 Generating random models connected from initial state

```
Output: Stochastic game SG where the initial state is connected to any $s \in S$
 1: Create S with a random $n \in \mathbb{N}$
 2: Partition S uniformly at random into $S < \square >$ and $S < \circ >$
 3: Enumerate $s \in S$ in any random order from 0 to n-1
 4: Set $s_0$ to the state with index 0
 5: for $s = 1 \to n - 1$ do      \* Forward Procedure * \
 6:     if s does have an incoming transition then Continue (Skip iteration)
 7:     else
 8:         Pick any state $s'$ with index smaller than s
 9:         Create an action a that starts at $s'$ Need to handle correct action indexing
10:         Assign to (s, a) a positive probability of reaching s
11:         Create a valid probability distribution for (s, a) by applying FillAction(s', a)
12:         Add a to $Av(s')$
13:         Add a to A
14: for $s = n - 1 \to 0$ do      \* Backward Procedure * \
15:     Pick a random number $m \in [M - |Av(s)|]$      \* Add as many actions as possible * \
16:     if $|Av(s)| = 0$ then $m \leftarrow \max\{m, 1\}$      \* Every state needs to have at least one action * \
17:     for $i = 1 \to m$ do
18:         $(s, a_i)$ = FillAction(s, $a_i$)
19:         Add $a_i$ to $Av(s)$
20:         Add a to A
```

we introduce a new action that we add to A. Also we add the state-action pair to Av. Thus any Av is function of $S \to 2^A$.

Next we need to prove that for every $s \in S$ and every action $a \in Av(s)$ the transition function $\delta(s, a)$ yields a probability distribution. In other words we need to validate that for every $s' \in S$ it holds that $\delta(s, a, s') \in [0, 1]$.

Whenever we introduce an action a to the set of enabled actions $Av(s)$ of a state $s \in S$, we have $\delta(s, a, s') = 0$ for all $s' \in S$. We then pick targets of (s, a) and increase their transition probability. Whenever we increase a transition probability, we increase it by a number $\in (0, 1]$. Furthermore, we never increase any transition more than once, except for Line 8 in Algorithm **??**. However, there we only increase the transition probability until for given state s and action a it holds that $\sum_{s' \in S} \delta(s, a, s') = 1$. Thus, no $\delta(s, a, s')$ can be above 1.

Algorithm 2 FillAction(s_{out}, a)

```
Input: outgoing state $s_{out}$, action $a$
Output: action $a$ that has a valid underlying transition probability distribution
1: repeat
2:     Pick a random state $s_{in}$ where $\delta(s_{out}, a, s_{in}) = 0$
3:     Increase $\delta(s_{out}, a, s_{in})$ by a random number $\in (0, 1]$
4: until $\forall s_{out} \in S : \delta(s_{out}, a, s_{in}) > 0$ or $\sum_{s_{out} \in S} \delta(s_{out}, a, s_{in}) \geq 1$
5: if $\sum_{s_{out} \in S} \delta(s_{out}, a, s_{in}) > 1$ then
6:     Decrease the most recently modified $\delta(s_{out}, a, s_{in})$ so $\sum_{s_{out} \in S} \delta(s_{out}, a, s_{in}) = 1$
7: else if $\sum_{s_{out} \in S} \delta(s_{out}, a, s_{in}) < 1$ then
8:     Increase the most recently modified $\delta(s_{out}, a, s_{in})$ so $\sum_{s_{out} \in S} \delta(s_{out}, a, s_{in}) = 1$
   return ($s_{out}$, $a$)
```

In Line 6 of Algorithm **??** we may also decrease $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}')$. However, this decrease can never yield a $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}') < 0$. This is because per definition of the loop in Algorithm **??**, $\sum_{\mathsf{s}_{in} \in \mathsf{S} \setminus \{\mathsf{s}'\}} \delta(\mathsf{s}, \mathsf{a}, \mathsf{s}_{in}) < 1$ and $\sum_{\mathsf{s}_{in} \in \mathsf{S}} \delta(\mathsf{s}, \mathsf{a}, \mathsf{s}_{in}) > 1$. Thus, if $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}')$ is decreased by Δ to guarantee equality of the sum to 1, it must hold that $\delta(\mathsf{s}, \mathsf{a}, \mathsf{s}') - \Delta \geq 0$.

In conclusion, Algorithm **??** generates formally correct stochastic games. □

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

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

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

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 **??**:

Basis : $i = 1$: The only state with a smaller index than s_1 is s_0. Since there must be a state with a smaller index that has an action with a positive transition probability to s_1, $\delta(\mathsf{s}_0, \mathsf{a}, \mathsf{s}_1) > 0$ and so the initial state is connected to s_1.

Step : $i \leftarrow i$: Again, 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 $\mathcal{SG} \subset \mathcal{SG}_{Algo}$:

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

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 $S < \square >$ and $S < \circ >$ is the same for SG and SG$'$.
3. SG and SG$'$ have the same initial state.
4. All state-action pairs in SG and SG$'$ yield the same probability distributions in δ.
5. Every state in SG and SG$'$ have the same actions.

SG and SG$'$ being the same means that there is an automorphism where SG can be mapped to SG$'$.

Since we decide randomly on the number of states, the initial state and the partitioning of S into $S < \square >$ and $S < \circ >$, there is trivially a run where statements 1, 2 and 3 hold.

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

To show that statement 5 holds, note that each SG $\in \mathcal{SG}$ 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 on 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 (s', a) can lead into arbitrary states and the probability distribution of (s', a) can be arbitrary, we can recreate the minimal set of state-action tuples in SG$'$.

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

Note that although $\mathcal{SG}_{Algo} = \mathcal{SG}$, in general Algorithm **??** does not sample $\mathcal{SG}$ 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 **??** favours 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. Adding Prefix-Graphs / Configurations

Since the purpose of random generated models is to sample from a space as big a possible, we have few insights on the structure of the model. Thus, this method is not very well suited if one wants to analyse behaviour 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. Therefor, we have added the option to create models via code. On one hand this allows for more influental parameters. On the other hand, we can prepend models to other models. Thus, we can for example prefend a model to every case study and see how much the added component influences algorithm performance. The prepended model contains the new initial state and instead of reaching a target it reaches the loaded model. Since the loaded model never reaches the prepended model, the runtime of any topological variant of an algorithm is the sum of the runtime to solve both models.

5. Implementing Random-Generated Graphs

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

5.1. Parameters and Guidelines for Model Construction

We have implemented a constrained version of Algorithm **??** 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 due and arbitrarily small transitions due to finite memory.

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 different on two models. For example is it 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 algorihtm performance on the models.

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 to use our default parameters. The parameters in Subsection 5.2 provide an overview that 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 models.

5.1.1. Limits and Additional Guideline-Options

Although the parameters we expose for random-generation can constrain some of the 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.

For this, we provide the option in our implementation to use other scripts for model generation that are then 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 root, but are not able to create any game in $\mathcal{SG}$.

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 atmost k children, where k is a user 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 not have 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 backwards process and are there to enable the generation of end components.

We refer to the guideline that controls the SECs of a model by *RandomSEC*. The procedure requires a minimal and maximal size boundary for all SCC $[a, b]$ and a number of overall states in the stochastic game n. First, we create subgames of size $[a, b]$. The subgames are created by the algorithm described in Section 4.3. We use then Tarjan's Algorithm for strongly connected components to identfy the SCCs of the created subgame. Next, we unify all the SCCs of the subgame by using the topological enumartion Tarjan's Algorithm provides. We connect the SCCs in a circular fashion along the enumartion, 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 consider the implementation details and provide guidance for using, understand and extending our implementation. The random generation is split into multiple python modules to maximize extendability and readability. All relevant python modules for this are located in the folder "random-generated-models". 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 state should the generated model have
- numModels: How many models to create
- numMinActions: probably deprecated after rewriting code
- guideline: Which guideline to use when generating the model. Will use by default Algorithm **??**.
- smallestProb: What is the smallest probability that is allowed to occur?
- backwardsProb: How probable is a state to have actions in the backwards procedure?
- branchingProb: How probable is that an action has multiple transitions
- maxBranchNum: How many positive transition-probabilities may one action have atmost?
- forceUnknown: Try to reduce the number of known states. This will violate the size parameters by introducing a new sink and target.

The generated model will be named after the parameter settings 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 supposed to be the interaction point for users.

The creation of the stochastic game is outsourced in another python module. The random generation as described in Algorithm **??** 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 guildeline - implemented in module sccGraphGenerator.py - inherit from graphGenerator.

The module "graphGenParams.py" contains a class that contains all the parameters any graph generator requires to function. I should probably rewrite this such that every graphGenerator has its own 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 on which states have how many actions or 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 which decides to which states the branching transitions of an action should connect. 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. During the generateGraph call. After generateGraph(), the GraphGenerator instance should hold all the information about the game in its class variables.

6. Algorithm Analysis

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

$\textbf{State} - \textbf{related}$:

- Number of states
- Number of targets*
- Number of sinks*
- Number of unknown states*
- Number of Maximizer states*
- Number of Minimizer states*

$\textbf{Actions} - \textbf{related}$:

- Number of maximum actions per state
- Number of maximum transitions per action
- Number of actions with probabilistic actions*
- Average number of actions per state

$\textbf{Transitions} - \textbf{related}$:

- Smallest occuring positive transition probability in the model
- Average number of transitions per action

$\textbf{MEC} - \textbf{related}$:

- Number of MECs

- Size of biggest and smallest MEC
- Average MEC-size and Median MEC-size

$\textbf{SCC} - \textbf{related}$:

- Number of SCCs
- Size of biggest and smallest SCC
- Average SCC-size and Median SCC-size
- Longest chain of SCCs from the initial-state SCC

The features marked with a * can be represented as relative values in relation to the number of states or actions respectively. Relative values are value useful since they are independent of the model size and thus make comparison easier. Furthermore, we analyse the shortest paths of the attractor [This would force me to introduce attractors only for this section...] from the initial state to any target.

6.1. 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 dataloading phase. To assert whether a value is correct, we need a reference value whose result we believe to be true. We usually use OVI or $\mathbf{SI_{LP}}^{\mathbf{T}}$ 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. For model-features we usually use 0 as replacement value. If an algorithm cannot 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 occuring 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.

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. Also, many visualization methods like heatmaps or statistical tests perform operations on the data. Thus, untrue values can falsify the correlation between features.

To handle this problem, in addition to the analysis of the whole data with penalty-values, we only consider data where we have a value for every feature.

7. Results

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

Regarding runtime a lot of stuff may change. Iterations wont change as long as we use the same deterministic algorithms, so its a more stable metric.

Various studies we have made, knit into a nice purple string and story. Here might appear: Maybe it would also be cool to show preformance differences in real case studies in comparison to the random generated models and make conclusions Ideally: We did previously not have enough models to see behaviour XYZ but now we can.

7.1. Experimental Setup

Various algorithms we consider were already implemented in PRISM-games [**prismgames3**]. We needed to extend PRISM-games by the algorihtms $\mathbf{SI_{LP}}$, $\mathbf{SI_{LP}}^{\mathbf{T}}$ and **TOPAlg**. Moreover, for **TOPAlg** we added precise Markov chain solving, which was not present in PRISM-games before and extended the strategy iteration (which was implemented in [**gandalf20**]) to use this precise solving. 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.

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. Most likely I may move all the features into the appendix and only highlight 5-10 cool features. What we can read from the box plot is this:

- Models have around 2 actions per state and 1.5 Transitions per action
- Some of our models were trivial
- A huge part of the models is actually trivial
- Generally, the number of states is evenly split to Maximizer and Minimizer
- Usually, around 70 to 85% of all actions are deterministic

By furthermore printing the maximal and minimal ocurring values of each feature we obtain that the smallest ocurring positive transition probability is 0.001.

With this information we can draw conclusions about which structural cases do not appear in the real case studies. None of the models contain these cases:

- Models with a large number of actions per state
- Models with a large number of Transitions per action
- Models with very small transition probabilities like 1e-6

We can also use boxplots to assess whether our random-generation procedure favours some feature-values over others or is uniformly distributed.

The insights we generate from this are:

- We have raised the number of unknowns on average to 30% in comparison to the 20% of the real data set. As seen from NumSinks, almost all known states are sinks. This is because we can have both trivial and non-trivial minimizer MECs that operate as sinks. We could try to remove them, but then we would intervene with the random nature of the procedure.
- As mentioned in Chapter 4, our procedure generates models with a tendency to having few actions per state and few transitions per action.
- The rest of the current image is very dependent on the parameters we set, so it pointless talking about it. Or we could use is as example. However, I should create 100 models with the same parameters and then see their feature distribution of non-set features.

7.3. Algorithm Analysis Results

In this section, we compare various prominent algorihtms and their optimizations on real case studies, handcrafted examples and our randomly generated models to

Overall, we compare the following algorithms:

- **VI**: Value iteration with naive stopping criterion
- **BVI**: Bounded value iteration providing a guarantee [Kel+18]
- **OVI**: Value iteration with upper-bound guessing [**What?**]
- **WP**: Bounded value iteration with the widest path approach for the upper bound [**widestPath**]
- $\mathbf{SI_{LP}}$: Strategy iteration with linear programming as MDP solution technique

Additionally, we use optimizations on these algorithms that we also evaluate in our benchmarks. These are:

- G: The Gauß-Seidel variant of value iteration for **BVI** and **OVI**.
- D: The idea from [**KKKW18**] to only deflate every 100 steps for **BVI**.
- T: The topological variant of **BVI**, **OVI** and $\mathbf{SI_{LP}}$.
- **TOPAlg**: The precise variant of topological value iteration.

Case studies

We consider case studies from three different sources: (i) all real case studies that were already used in [**gandalf20**], which are mainly from the PRISM benchmark suite [**PRISMben**]. We omit models that are already solved by pre-computations. (ii) several handcrafted corner case models: haddad-monmege (an adversarial 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 [**gandalf**]. (iii) randomly generated models generated by Algorithm **??** and our additional guidelines from Subsection 5.1.

7.3.1. Comparing Algorithm-Performance

First, we provide a general overview of the performance of all algorithms on our benchmarking set. We do this by providing a fancy graph that I have copied from Maxi. I should also steal his description

Should make the both plots in jupyter next to each other. This gives best quality screenshots. If not possible then at least make the height smaller

We also should do the same thing with iterations!

What we can read from the graphs:

- $\mathbf{SI_{LP}}^\mathbf{T}$ is accumulated better than $\mathbf{SI_{LP}}$.

- $\mathbf{SI_{LP}}^{\mathbf{T}}$ is very good.
- $\mathbf{SI_{LP}}$ good on random but not so good on real. Why?
- **WP** is usually the best value iteration approach
- **OVI** is usually better than **BVI** for small models
- *TOP* does not seem very promising. We will find out why in the scatter plots
- The optimizations do not seem to do much except for the topological switch for $\mathbf{SI_{LP}}^{\mathbf{T}}$ and **WP**
- Why GBVI is not always better than BVI (iterationwise)

We investigate these clues by using more specific graphs.

Topological extension for strategy iteration with linear programming is usually benefitial

Show Scatter. Should usually be better. Maybe would be cool to do the same for OVI and BVI and show that there this is not the case. However, for that we may also point to GANDALF

Topological strategy iteration with linear programming performs is good

Altough value iteration is usually regarded to be the most performant algorithm type for solving stochastic games, $\mathbf{SI_{LP}}^{\mathbf{T}}$ yielded the 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.

Although linear programming is believed to perform worse than value iteration for MDPs [**ANYTHING?**], we could not find clear superiority of value iteration on the reachability benchmarks for MDPs provided on [**QComps**].

Lastly, it is believed that linear programming scales worse than value iteration for huge models. But even if that is true, $\mathbf{SI_{LP}}^{\mathbf{T}}$ may be a good complementary solution approach in case a model is especially hard for value iteration. Also, the topological improvement allows to solve models with huge numbers of SCCs faster than value iteration. And also value iteration was the focus of research for the last 20 years. It is very likely that LP could be improved. Atm we do not even deflate but use MIP to encode the maximum-best-exit-constraints. But this should maybe go into future work

Comparing OVI and BVI

We use scatter plots, where every point describes the runtime of BVI in relation to the runtime of OVI to solve the model.

Show here OVI and BVI compared. Iterations and Runtime. And also Real and Random. Its a 2x2 Graph. May also overlay random and real

Clearly, OVI performs time-wise better than BVI. This does not hold true for the iterations. However, the most time consuming part of OVI is the deflation, of which OVI requires less. However, for big models an additional failed verification phase can be detremental for OVI. We should be able to see this in the big models. So I would probably not even recommend it. This also shows the need for big models + BVI is an anytime approach

Comparion Vanilla to Optimization Switches

No big difference. Enter into the scatter plot both at the same time

Problems with TOP

TOP seems to perform worse than vanilla in general. This is because we solve the DTMCs with exact methods, requiring a matrix inversion, which is $\mathcal{O}(n^2)$-operation, where n is the number of states in the Markov Chain. However, using iterative methods to solve the DTMC may yield better results. We have also tried MDP-LP-solving instead of MDP-SI-solving, but it was still worse than $\mathbf{SI_{LP}}^{\mathbf{T}}$.

Gauss-Seidel BVI

Generally less iterations, but matrix operations of vanilla is faster. Can sometimes be worse due to unlucky SCCs based on lower bound. Then the upperbound can fall behind the vanilla-upperbound GS-bvi along topological enumeration of states could be better, check the results...

7.3.2. Searching Correlations between Algorithms and Features

Next, we are interested in correlations between algorithm-performance and values of features. Ideally, we would like to find cases where one algorithm scales better with certain features than others. This would allow to analyse the graph structure and decide based on the features we get which algorithm is most likely the best one to use in this case. To find these correlations, we use the following visualization tools:

Heatmaps Heatmaps visualize correlation matrices - matrices where one feature is mapped against another. The higher the correlation value, the stronger a correlation between two features is. On the diagonal of the matrix the correlation is maximal since there is a directly proportional correlation between a feature and itself. Ideally, we would like to get clues from the heatmap which features or correlations we should investigate. However, for the most part we could not gather any non-trivial information from heatmaps.

Scatter Plots We plot algorithm runtime / iterations against feature values. Each point corresponds to the feature-value and runtime of a model.
What is there to find in these graphs?

- Clearly visible how TOP scales with SCC-size
- All scale with number of unknowns, which is no surprise but might be mentioned.

One Dimensional Graphs We can get models where proposition A is fulfilled and compare them to models where A is not fulfilled. We can plot the models where A is true against models where A is false to see whether they hold structural differences. This did not seem to be the case in general.

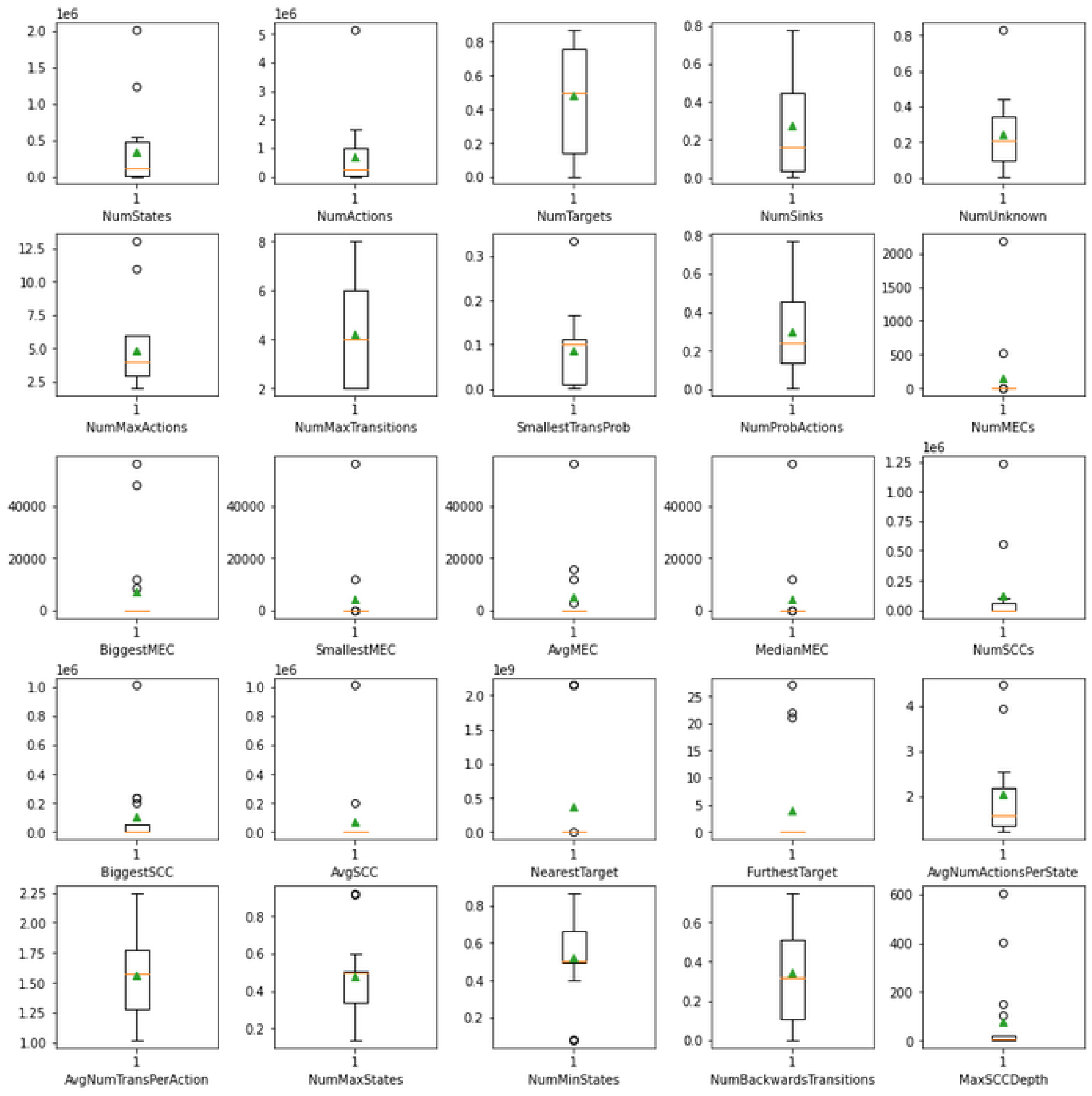


Figure 7.1.: Box-plot of the feature distribution of the real case-studies. In each graph, the distribution of the values of one separate feature over all real case studies is visualized. The orange line marks the median of a feature over all models and the green triangle marks the average. The bounds of the boxes mark the 25 and 75 percentile, and the lines extended by the whiskers mark the 10 and 90 percentile. Not sure about the 10 and 90 percentiles Dots outside of whiskers represent outliers. The relative number of probabilistic actions for example is between around 15% and 45% in half of the models. In half of the models, more than 20% of actions are probabilistic.

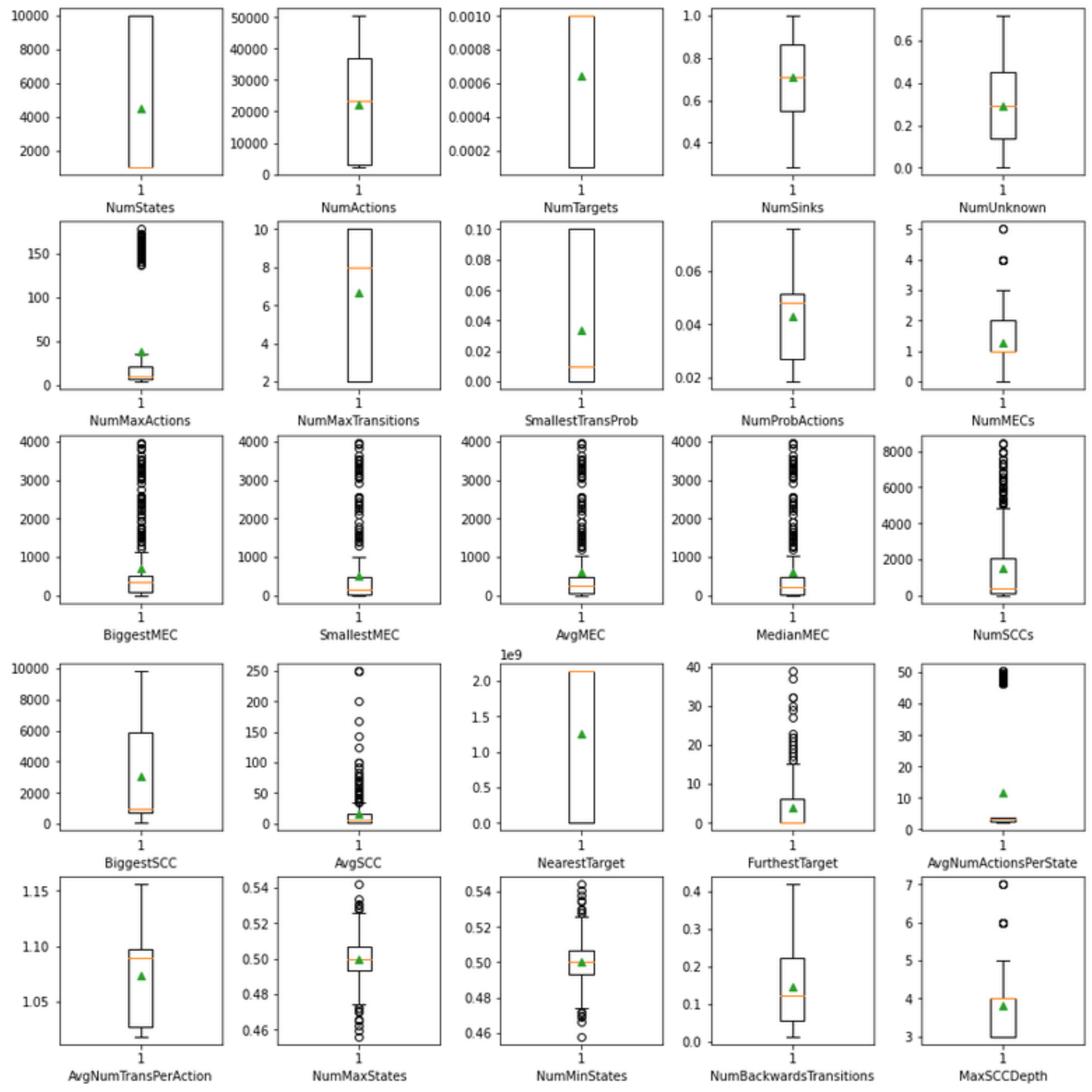
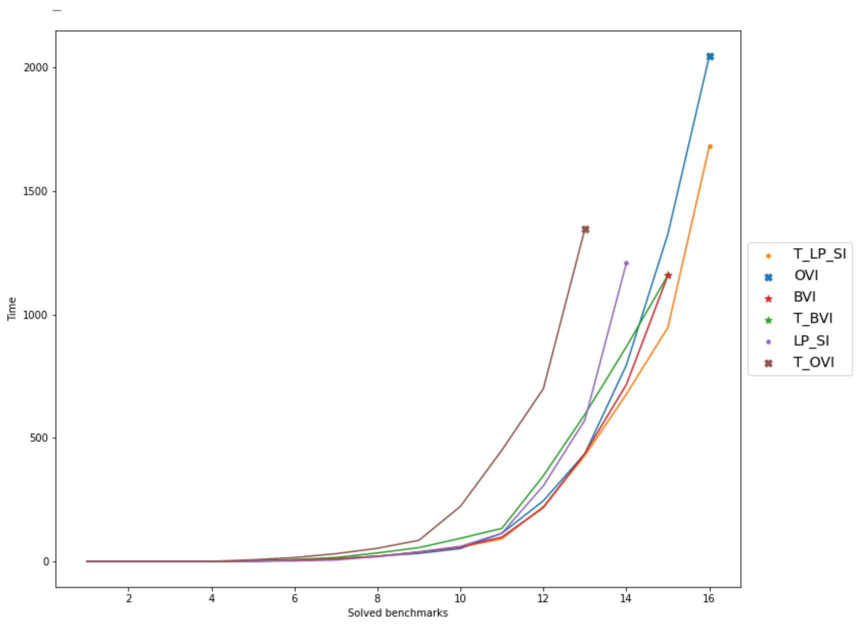
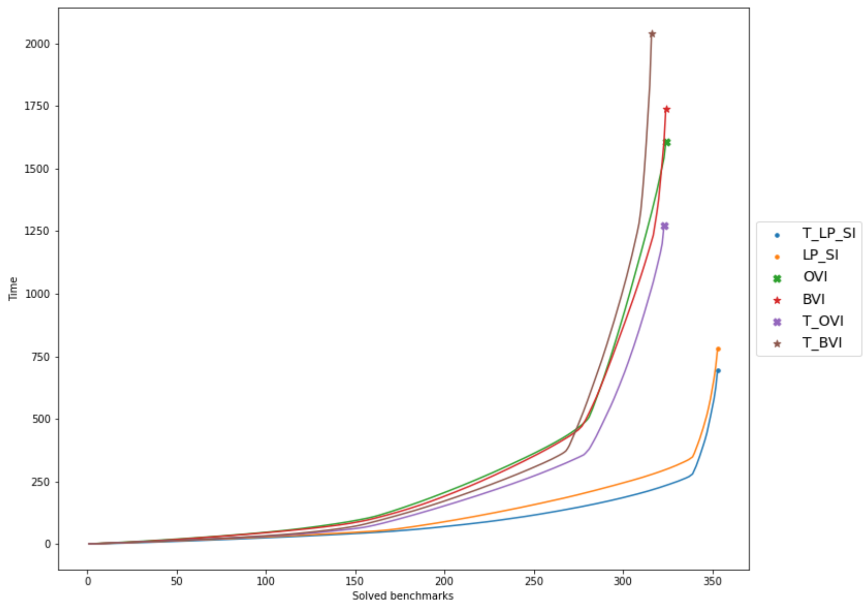


Figure 7.2.: Feature Distribution of random models



(a) Performance Overview on real case studies

```
Conf T_LP_SI:   694.389000       Not Finished / Error / Wrong Result:    0
Conf LP_SI:     779.761000       Not Finished / Error / Wrong Result:    0
Conf OVI:       1605.724000      Not Finished / Error / Wrong Result:    29
Conf BVI:       1738.781000      Not Finished / Error / Wrong Result:    29
Conf T_OVI:     1271.406000      Not Finished / Error / Wrong Result:    30
Conf T_BVI:     2041.246000      Not Finished / Error / Wrong Result:    37
```



(b) Performance Overview on random generated models

Figure 7.3.: Overview of Algorithm Performance

8. Conclusion And Future Work

General outline: We have found some stuff that is cool and we have guided a so far unexplored way of analysing Data

A huge benefit of including data analysis is that we could simply protoype an algorithm idea and assess very fast where on the spectrum of performance it lies. Also we can simply add more algorithms and still have a good overview, maybe an even more solid overview. This is not the case with the tables. Probably we would not even have tried $\mathbf{SI_{LP}}^{\mathbf{T}}$ is we didnt have the visualization tools.

We have shown again that structure is very important for algo performance.

We have created an arbitrarily big or small benchmarking-suite for stochastic games that could be profitable for the whole stochastic game community. With the random models and our analysis tool we cool derive some interesting property of algorithm X and found out Y about algorithm Z.

8.1. Future Work

The features we track are all very basic, but it is a good step forward. However, we still cannot explain a lot of what is going on. For that we may need new structural concepts and features we track.

I believe that if someone would go further down this direction, a portfolio solver is definitely possible and maybe the best solution for users.

PRISM cannot handle explicit big files. A next step would be to create an optimization system to create random files that hold chunks of the data in implicit blocks to make the files easier parseable.

The analysis techniques could be improved by including stochastic tests and artificial intelligence algos to cluster features and find correlations. This could then even allow for a portfolio solver which decides based on machine learning which algorithm to apply based on the graph structure.

A. Appendix: Additional benchmarks

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.

A.1. 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.

A.2. More Formal Random Generation Algorithm

Basically, replace every function Im speaking about with a tuple (which it formally is) and do set-operations. Not as nice in my opinion as a descriptive approach. This is not done yet. Also I will need a formal fill actions. Whenever you add an action with a transition distribution that sums up to one, every state NOT reached must be added with transition probability 0.

Algorithm 3 Generating random models connected from initial state

```
Output: Stochastic game SG where the initial state is connected to any $\mathsf{s} \in \mathsf{S}$
 1: Create S with a random $n \in \mathbb{N}$
 2: Partition S uniformly at random into $\mathsf{S} < \Box >$ and $\mathsf{S} < \circ >$
 3: Enumerate $\mathsf{s} \in \mathsf{S}$ in any random order from 0 to n-1
 4: Set $\mathsf{s}_0$ to the state with index 0
 5: $\delta = \emptyset$
 6: $\mathsf{A} = \emptyset$
 7: for $\mathsf{s} = 0 \rightarrow n-1$ do $\mathsf{Av}(\mathsf{s}) = \emptyset$
 8: for $\mathsf{s} = 1 \rightarrow n-1$ do      \* Forward Procedure * \
 9:     if $\exists \mathsf{s}' \in \mathsf{S}, \mathsf{a} \in \mathsf{A} : \delta(\mathsf{s}', \mathsf{a}, \mathsf{s}) > 0$ then Skip state
10:     else
11:         Pick any state $\mathsf{s}'$ with index smaller than s
12:         Pick a random $p \in (0,1]$
13:         Create a new action a
14:         $\mathsf{A} \leftarrow \mathsf{A} \cup \{action\}$
15:         $\delta \leftarrow \delta \cup \{(\mathsf{s}', \mathsf{a}, \mathsf{s}, p)\}$
16:         Make a a valid action by applying FillAction(s', a)
17:         $\mathsf{Av}(\mathsf{s}') \leftarrow \mathsf{Av}(\mathsf{s}') \cup \{action\}$
18: for $\mathsf{s} = n-1 \rightarrow 0$ do      \* Backward Procedure * \
19:     Pick a random number $m \in [M - |\mathsf{Av}(\mathsf{s})]$      \* Add as many actions as possible * \
20:     if $|\mathsf{Av}(\mathsf{s})| = 0$ then $m \leftarrow \max\{m, 1\}$      \* Every state needs to have at least one action * \
21:     for $i = 1 \rightarrow m$ do
22:         $(\mathsf{s}, \mathsf{a}_i)$ = FillAction(s, $\mathsf{a}_i$)
23:         Add $\mathsf{a}_i$ to $\mathsf{Av}(\mathsf{s})$
24:         Add a to A
```