Chair of Software Modeling and Verification (Computer Science 2)

RWTH AACHEN UNIVERSITY

This thesis was submitted at the Chair of Software Modeling and Verification.

AN IMPLEMENTATION OF A DIRECT REDUCTION FROM PARITY TO REACHABILITY ON STOCHASTIC GAMES

Merlin Weise
Student ID: 444922
RWTH Aachen University
2025

Bachelor's Thesis

First Examiner:	Univ.-Prof. Dr. Ir. Dr. h.c. Joost-Pieter Katoen
Second Examiner:	apl.-Prof. Dr. Thomas Noll
Supervisors:	Alexander Bork & Dr. Raphaël Berthon
Submission Date:	August 9, 2025

An Implementation of a Direct Reduction from Parity to Reachability on Stochastic Games

Merlin Weise

Abstract

In this thesis, we study *stochastic parity games* (SPGs), which extend classical parity games by incorporating probabilistic behavior. They model complex systems with uncertainty and adversarial behavior, enabling formal verification. Since SPGs are less well supported than *simple stochastic games* (SSGs), we present a tool called Stargate that transforms SPGs into SSGs and then solves them using existing tools such as PRISM-games. We define specification languages for SPGs and SSGs and implement a reduction based on the work of Berthon et al. [3], which we extend by an ε-optimality concept. Furthermore, we describe a transformation from SSGs to *stochastic multiplayer games* (SMGs) to enable compatibility with PRISM-games. We validate Stargate on a set of custom benchmarks regarding correctness, runtime, and solving strategies.

Keywords: Stochastic Parity Games, Simple Stochastic Games, Reduction, Model Checking, PRISM, Algorithm Implementation

RWTH Aachen University
2025

Contents

1 Introduction

In this thesis, we study *stochastic parity games* (SPGs), a class of games that extend classical parity games by incorporating probabilistic elements.
Stochastic games enable the modeling of complex systems with inherent uncertainty and play a key role in the verification of probabilistic systems with adversarial components, such as network protocols or multi-agent systems [21]. They are particularly relevant for automated planning and formal verification, as they capture both probabilistic behavior and nondeterministic adversarial choices.
Stochastic games are deeply related to *Markov chains (MCs)* and *Markov decision processes (MDPs)*, as they generalize these models by combining probabilistic behavior with strategic choices made by multiple players. Both Markov chains and Markov decision processes play a central role in the modeling and verification of systems with uncertainty [2], e.g., discrete time stochastic hybrid systems [12].

Motivating Example: Consider a communication protocol in which requests must eventually be acknowledged. An adversarial environment may delay or drop messages, and probabilistic failures may occur due to unreliable channels. To guarantee responsiveness, the system must ensure that "every request is eventually acknowledged" regardless of these uncertainties. This property can be expressed as a fairness condition and formalized as a parity objective. SPGs allow reasoning about such behavior by modeling the interaction between the system, the adversary, and probabilistic events.

A parity condition can represent any so-called ω-regular property, including safety and fairness objectives [23]. Moreover, SPGs pose interesting theoretical challenges by combining optimization under uncertainty with complex winning conditions. Hence, SPGs are both practically relevant and of significant theoretical interest.
Simple stochastic games (SSGs), also referred to as *stochastic reachability games*, concern reachability objectives in game graphs. SSGs are widespread systems and deeply related to SPGs. Moreover, SSGs are currently better understood in the field of probabilistic model checking.

Our goal is to implement a STochastic pARity GAme TransformEr that we call Stargate. The code of Stargate is at https://github.com/merlinweise/stargate. The key aspect of Stargate is to solve SPGs. Since solving SSGs is more supported [21], we transform SPGs into SSGs following [3] and then solve these with existing SSG solvers.

An overview of Stargate is given in Figure 1. Starting from the bottom left with the specifications for SPGs and SSGs we move over to the transformation methods until we reach the tool PRISM-games.
The solving process of an SPG can be summarized in the following steps:

1. Reading an SPG specified in SPLANG (described in Section 4.1)
2. Transforming SPG to SSG (described in Section 5.2)
3. Transforming SSG to SMG (described in Section 6.2)
4. Solving SMG in PRISM-games [21] (described in Section 6.3)

First in Section 3, we clarify the notation used in this paper.

Specifying SPGs and SSGs: For the purpose of reading in SPGs and SSGs, in Section 4 we discuss the specification languages SPLANG and SSLANG for SPGs and SSGs, respectively. We defined them to enable a user-friendly way to input SPGs and SSGs. These follow a structure similar to *stochastic multiplayer game* (SMG) specifications for PRISM-games.

Reducing SPGs to SSGs: In Section 5, we discuss the reduction from SPGs to SSGs presented by Berthon et al. in [3]. The reduction involves applying a gadget construction on the SPG. In [3], only a reduction for optimal strategies was introduced. We extend this concept by adding a variant with ε-optimality, i.e., allowing the result to differ from the optimal result by an amount determined by ε that is set by the user. This prevents values in the gadget from assuming extraordinarily small values, since small values lead to limitations in the solving process.
Afterwards, we shed light on the implementation in Stargate of this reduction.

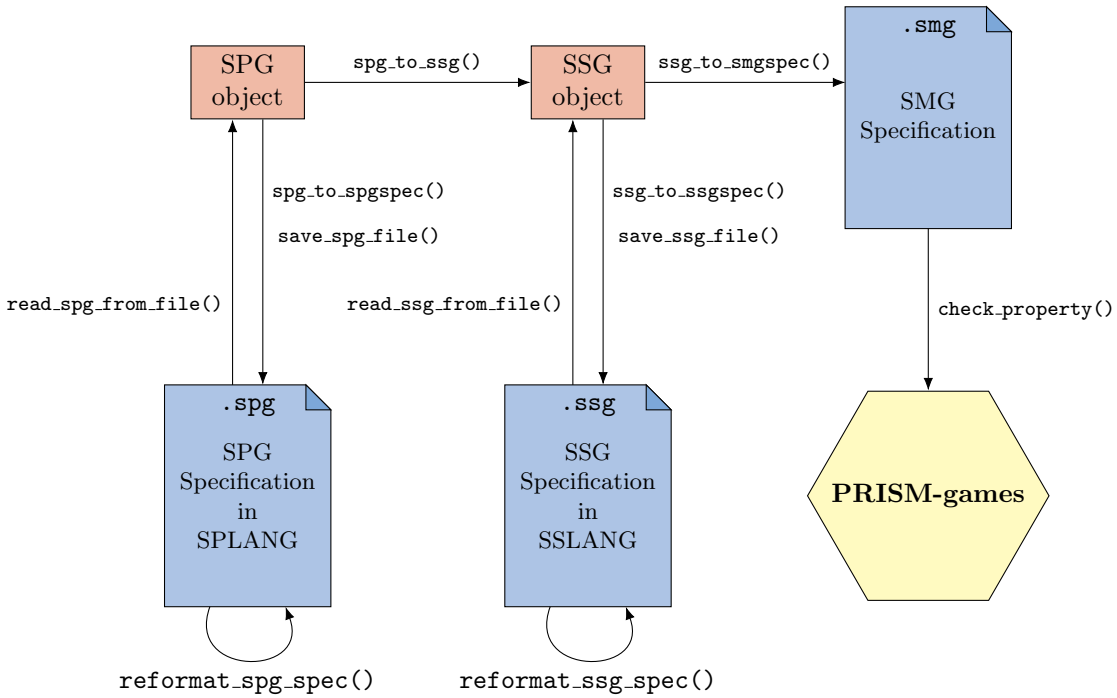


Figure 1: Overview of STARGATE showing models represented as objects and files and respective transformation methods. The notations mean the following:
Stochastic Game files, Stochastic Game objects, PRISM-games tool, `transformation methods`

Solving SSGs with PRISM-games: To the best of our knowledge, apart from PRISM-games there are no other full-featured solvers for SSGs. In order to solve the SSG, we use an extension of PRISM-games [18] which we discuss in Section 6.1. Since PRISM-games does not natively support SSLANG, we need to convert the SSG into a format PRISM-games can process. Therefore, in Section 6.2 we introduce a transformation from SSGs to SMGs. Since in PRISM-games SMGs have a different underlying concept of state space compared to our understanding of SSGs, this transformation is not trivial. Thus, we develop three approaches of the algorithm creating an SMG from a given SSG. The first two approaches modify the SSG first and then transform it while the third approach directly creates the SMG and adds synchronization.
Furthermore, in Section 6.3 we discuss the solving procedure for the resulting SMG in PRISM-games. We implement several methods that use PRISM-games as a subprocedure in order to analyze the SMG, and create a graph representation for illustration purposes.

Validation: We discuss the validation of STARGATE in Section 7. We first compare the three versions to transform an SSG into an SMG presented in Section 6.2. Second, we test STARGATE regarding the correctness of the results upon several self-created benchmark SPGs. Third, we analyze the runtime, size complexity, and different solving algorithms using a self-created benchmark set.

We found that STARGATE computes correct results for smaller models. The results become more imprecise for moderately large models. We suspect these imprecisions to mainly originate from the current state of PRISM-games.
Concerning runtime and space resources, STARGATE has an acceptable scalability even for larger models.

Finally, in Section 8, we discuss the value and role of STARGATE.

2 Related Work

In this section, we review previous work related to the analysis of simple stochastic games and stochastic parity games.

Simple Stochastic Games: SSGs were introduced in 1953 by Shapley [25]. Condon showed that the associated decision problem of solving an SSG, i.e., deciding if the probability for a player to win is greater than $\frac{1}{2}$, lies in NP ∩ co-NP [11] .

Stochastic Parity Games: Parity objectives are a canonical way to represent *ω-regular* properties in infinite-duration games, providing a uniform framework for expressing complex specifications such as fairness, liveness, and safety [14]. The most relevant work for this thesis is the paper by Berthon et al. [3], which presents a direct polynomial reduction from SPGs to SSGs. We examine this reduction in detail in Section 5.
Parity objectives in the context of stochastic games, which we address as SPGs, have been studied in various other works. [4] addresses ω-regular objectives under incomplete information. For this purpose, an algorithm to solve SPGs with EXPTIME complexity was introduced. Chatterjee et al. [6] present a strategy improvement algorithm, which constitutes the first non-exhaustive method for solving SPGs, and a randomized subexponential algorithm that solves the SPG.
Another approach [16] uses model-free reinforcement learning on SPGs that were reduced to a family of SSGs. The reduction is similar to the one presented in [3] using a gadget.

Solving Stochastic Parity Games: Apart from [3], several studies address solving SPGs through a direct reduction: GIST [7] is a tool that reduces SPGs to deterministic parity games and then solves these to deduce values for the SPG. PGSolver [13] is a tool offering several state-of-the-art algorithms for solving deterministic parity games.
[1] presents several polynomial reductions between different types of stochastic games, including a reduction from SPGs to mean-payoff games to discounted-payoff games to SSGs.
For solving SSGs, we consider the solver PRISM-games, first introduced in [9]. PRISM-games is an extension of the widely used model checker PRISM [20] that solves SMGs.

Simple Stochastic Game Solving Algorithms: There are multiple algorithms to solve SSGs. Křetínský et al. discuss in [18] Value Iteration which iteratively computes the values for each vertex to be optimal in the limit, and Policy Iteration which iteratively improves the strategies for the players which become optimal in the limit. Condon [10] presents two approaches: solving the SSG with a quadratic program with linear constraints and solving the SSG by using linear programming.

3 Preliminaries

For the following sections, we fix the used notation following [2, 3]:

Mathematical Foundation:

- $\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of *natural numbers*.
- Let r be a regular expression. $r^\omega = rrr\cdots$ denotes the infinite repetition of r.
- Let A be a countable set. $\mu : A \to \mathbb{Q}_{\geq 0}$ is a *(discrete) probability distribution* if and only if $\sum_{a \in A} \mu(a) = 1$. $\mathscr{D}(A) = \{\mu \mid \mu \text{ is a probability distribution over domain } A\}$ denotes the *set of all probability distributions* over A.
  The *support* of a probability distribution $\mu : A \to \mathbb{Q}_{\geq 0}$ is $supp(\mu) = \{a \in A \mid \mu(a) > 0\}$. When we define a probability distribution $\mu : A \to \mathbb{Q}_{\geq 0}$ only for $A' \subseteq A$, we assume that $\mu(a) = 0$, for all $a \in A \setminus A'$.

3.1 Discrete-Time Markov Chains

We introduce discrete-time Markov chains. These are basic models for probabilistic model checking. Moreover, they are fundamental for solving stochastic games, because a stochastic game with fixed strategies for both players can be represented as a Markov chain.

Definition 3.1.1: Discrete-Time Markov Chains

A *discrete-time Markov chain (MC)* is a tuple (V, δ, v_I) where

- V is a finite set of states,
- $\delta : V \to \mathscr{D}(V)$ is a probabilistic transition function, and
- $v_I \in V$ is the initial state.

From $v, v' \in V$ and $\delta(v)(v') = p, p > 0$, we define the transition from v to v' with probability p, and in short notation we use $\delta(v, v') = p$. For $X \subseteq V, v \in V$, we set $\delta(v, X) := \sum_{x \in X} \delta(v, x)$.

Paths serve in the analysis of MCs concerning reachability probabilities.

Definition 3.1.2: Paths

An *infinite path* through an MC is an infinite sequence $\pi = v_0 v_1 \cdots \in V^\omega$ with $\delta(v_i, v_{i+1}) > 0$ and $i \in \mathbb{N}$.
For $v \in V$, $Paths(v)$ denotes all infinite paths starting at v.

A *prefix* $\overset{*}{\pi} = v_0 \cdots v_i, i \in \mathbb{N}$ of an infinite path $\pi = v_0 v_1 \cdots \in V^\omega$ is called a *finite path* $\overset{*}{\pi} \in V^*$. Analogously, for $v \in V$, $Paths^*(v)$ denotes all finite paths starting at v.
For a path π, $inf(\pi) = \{v \in V \mid \forall n \in \mathbb{N}, \exists k \in \mathbb{N} : v_{n+k} = v\}$ denotes the set of infinitely often visited states. The probability of a finite path $\overset{*}{\pi} = v_0 \cdots v_n \in V^*$ is calculated by $\prod_{i \in \{0,\ldots,n-1\}} \delta(v_i, v_i + 1)$.

Definition 3.1.3: Reachability Probabilities

Let $M = (V, \delta, v_I)$ be an MC. Given a starting state v_0 and a set of target states $T \subseteq V$, we define the event of reaching T as $Reach(T) = \{v_0 v_1 \cdots \in V^\omega \mid \exists i \in \mathbb{N}, v_i \in T\}$. The probability of reaching T with starting state v_0 is $\Pr^{v_0}(Reach(T)) = \Pr(\{\overset{*}{\pi} \mid \overset{*}{\pi} \in Paths^*(v_0) \cap ((V \setminus T)^* T)\})$.

3.2 Stochastic Games

Stochastic Games are models to represent games with multiple players, winning objectives, and adversarial and random behaviors. These models are of central significance to this thesis.

Definition 3.2.1: Stochastic Arenas (derived from [18])

A *stochastic arena* is a tuple $(V, V_\exists, V_\forall, v_I, A, Av, \delta)$ where

- V is a finite set of vertices with $V_\exists \cup V_\forall = V$ and $V_\exists \cap V_\forall = \emptyset$
- $V_\exists \subseteq V$ is a set of vertices controlled by player Eve
- $V_\forall \subseteq V$ is a set of vertices controlled by player Adam
- $v_I \in V$ is the initial vertex
- A is a finite set of actions
- $Av : V \to 2^A$ assigns a set of available actions to all vertices
- $\delta : V \times A \to \mathscr{D}(V)$ is a transition function, that returns for a given vertex $v \in V$ and an available action $a \in Av(v)$ a probability distribution over successor vertices.

With $v \in V, a \in Av(v)$, we write $\mu_{v,a}$ for $\delta(v, a)$. Without loss of generality, we assume each vertex has at least one successor, i.e., each vertex is *non-blocking*. A *Markov decision process* (MDP) [24] can be modeled as a stochastic arena with one player, i.e., either $V_\exists = \emptyset$ or $V_\forall = \emptyset$. When for all $v \in V$ it holds that $|Av(v)| \leq 1$, the arena describes an MC.

Occasionally, we use a different notation for transitions in stochastic arenas. This facilitates addressing the transition function, as we alternatively represent them as tuples. Note that this representation differs from the representation of transitions in MCs.
For a stochastic arena $G = (V, V_\exists, V_\forall, v_I, A, Av, \delta)$ we call $t = (v, a, \mu)$ a *transition* of G if $v \in V, a \in Av(v), \delta(v, a) = \mu$. We denote $\mathcal{T}(G)$ as the set of transitions in the arena G. Let $e : V \times A \to 2^V, e : (v, a) \mapsto supp(\delta(v, a)), a \in Av(v)$. For $v \in V$, $a \in Av(v)$ and $e(v, a) = E = \{v_{t,1}, \ldots, v_{t,|E|}\}$ we say that there is a *transition* from v to E with action a. In this context, we call E the set of *end vertices* of this transition. For $t = (v, a, \mu)$, $e(t)$ denotes $e(v, a)$.
For $|E| = 1$ we say that this transition is *deterministic* and there is a transition from v to v' (instead of $\{v'\}$) with action a. For $|E| > 1$ we call this transition *probabilistic*.
For $t = (v, a, \mu)$, if $e(v, a) = \{v\}$, we call t a *self-loop*.

Definition 3.2.2: Strategies

Let $G = (V, V_\exists, V_\forall, v_I, A, Av, \delta)$ be a stochastic arena. A *(pure memoryless) strategy* σ of Eve is a function $\sigma : V_\exists \to A$ such that for all $v \in V_\exists, \sigma(v) = a, a \in Av(v)$.
A strategy γ of Adam is defined analogously.

For all $\sigma(v) = a$ with $v \in V_\exists, a \in Av(v)$, it follows that there is a transition from vertex v with action a. We write $\Sigma_\exists$ and $\Sigma_\forall$ for the set of all strategies for Eve and Adam, respectively.
For a fixed stochastic arena G, strategies σ and γ for Eve and Adam, we induce the MC $M_{\sigma,\gamma} = (V', \delta', v_I')$ with

- $V' = V$,
- $\delta'(u, v) = \delta(u, \sigma(u))(v)$ for $u \in V_\exists$,
- $\delta'(u, v) = \delta(u, \gamma(u))(v)$ for $u \in V_\forall$, and
- $v_I' = v_I$.

A *play* in an arena G is an infinite sequence of vertices $\pi = v_0 v_1 \cdots \in V^\omega$ where for all $i \in \mathbb{N}$ there exists $a \in Av(v_i)$ such that there is a transition $t = (v_i, a, \mu)$ where v_{i+1} is an end vertex. We denote by Π_G the set of plays, or simply Π when the respective stochastic arena is clear.

Definition 3.2.3: Winning Objectives

A *winning objective* for Eve is defined as the set $\Phi \subseteq \Pi$. As stochastic arenas model games where for every play there is exactly one winner, Adam's winning objective is $\Pi \setminus \Phi$. A play φ *satisfies* an objective Φ if $\varphi \in \Phi$.

We consider two types of winning objectives:

1. **Parity Objective:** We define a *priority function* $p : V \to \mathbb{N}$. For $v \in V, p(v)$ is called the *priority* of v. For a set of vertices $T \subseteq V$ we denote $\{p(t) \mid t \in T\}$ as $p(T)$.
   A *parity objective* is $PA(p) = \{\pi = v_0 v_1 \cdots \in \Pi \mid \min_{v \in inf(\pi)} p(v) \equiv 0 \mod 2\}$. Intuitively, a parity objective holds in a play if the smallest infinitely often seen priority is even.

2. **Reachability Objective:** Let $T \subseteq V$ be a set of target vertices. A *reachability objective* is $RE(T) = \{v_0 v_1 \cdots \in \Pi \mid \exists k \in \mathbb{N}, v_k \in T\}$.
   Intuitively, a reachability objective holds in a play if a target vertex is eventually reached.

Definition 3.2.4: Stochastic Games

Let G be a stochastic arena and Φ a winning objective. Then we call (G, Φ) a *stochastic game (SG)*.

If (G, Φ) is an SG and Φ is a parity objective, we call (G, Φ) a *stochastic parity game (SPG)*. In an SPG, unless stated otherwise, we assume that Eve has to satisfy a parity objective and Adam has the dual objective $\Pi_G \setminus \Phi$ meaning he tries to achieve a play where the smallest priority visited is odd.

If (G, Φ) is an SG and Φ is a reachability objective, we call (G, Φ) a *simple stochastic game (SSG)*. SSGs are also referred to as *stochastic reachability games (SRG)*. In an SSG, unless stated otherwise, we assume that Eve has the target reachability objective and Adam has the dual objective $\Pi_G \setminus \Phi$ meaning he tries to achieve a play where a target vertex is never reached.

Definition 3.2.5: Strategies in Stochastic Games

Let $SG = (G, \Phi)$ be an SG, and let $\sigma \in \Sigma_\exists, \gamma \in \Sigma_\forall$ be strategies in SG for Eve and Adam, respectively. Assume that Eve has Φ and Adam has $\Pi \setminus \Phi$ as a winning objective.
The probability measure $\mathbb{P}^v_{\sigma,\gamma} : 2^\Pi \to \mathbb{Q}_{\geq 0}$ assigns probabilities of achieving a play $\pi \in \Pi$, where $v \in V$ is the starting vertex of the play π.
For a starting vertex $v \in V$ in SG, we denote $\mathbb{P}^v_{\sigma,\gamma}(\Phi)$ as the probability that a play π, resulting from both players following their respective strategy, is winning for Eve with $\pi \in \Pi$. Analogously, Adam wins with probability $\mathbb{P}^v_{\sigma,\gamma}(\Pi \setminus \Phi) = 1 - \mathbb{P}^v_{\sigma,\gamma}(\Phi)$.

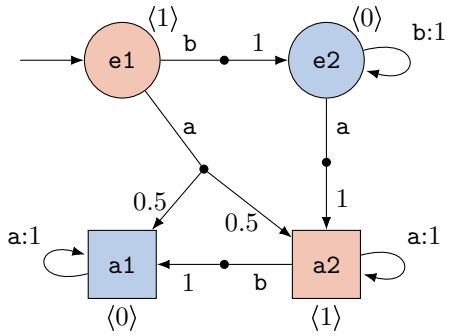
In our representation of SGs, the starting vertex of a strategy is in most cases the initial vertex of the SG's arena. In this case we write $\mathbb{P}_{\sigma,\gamma}(\Phi)$ and $\mathbb{P}_{\sigma,\gamma}(\Pi \setminus \Phi)$ for Eve's and Adam's winning probabilities, respectively.
The *value* of a vertex $v \in V$ is defined with a *value function* $\langle\exists\rangle(\Phi)(v) = \sup_{\sigma \in \Sigma_\exists} \inf_{\gamma \in \Sigma_\forall} \mathbb{P}^v_{\sigma,\gamma}(\Phi)$.
We occasionally denote the values of an SG with vertices V as a vector $\mathbf{v}$.
Intuitively, $\langle\exists\rangle(\Phi)(v)$ describes the probability of Eve achieving a winning play $\varphi \in \Phi$ while Adam tries to prevent it. Analogously, $\langle\forall\rangle(\Phi)(v) = \sup_{\gamma \in \Sigma_\forall} \inf_{\sigma \in \Sigma_\exists} \mathbb{P}^v_{\sigma,\gamma}(\Pi \setminus \Phi)$ describes the probability of Adam achieving a winning play $\pi \in \Pi \setminus \Phi$ while Eve tries to prevent it.
A strategy $\sigma \in \Sigma_\exists$ is optimal for Eve in v if $\inf_{\gamma \in \Sigma_\forall} \mathbb{P}^v_{\sigma,\gamma}(\Phi) = \langle\exists\rangle(\Phi)(v)$. Similarly, a strategy γ is optimal for Adam in v if $\inf_{\sigma \in \Sigma_\exists} \mathbb{P}^v_{\sigma,\gamma}(\Pi \setminus \Phi) = \langle\forall\rangle(\Pi \setminus \Phi)(v)$.
Pure memoryless determinacy is the property of an SG which describes that both Eve and Adam have optimal pure memoryless strategies. This property holds for all SPGs and SSGs [22].

We consider the following methods of solving SGs:

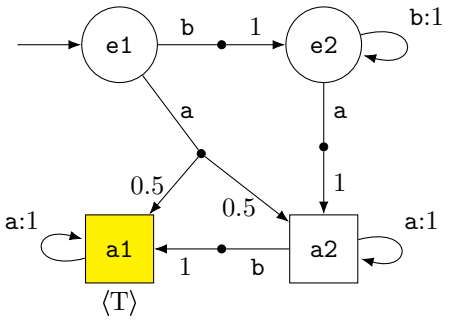
Definition 3.2.6: Solving Methods

1. *Qualitative Solving* means finding all vertices in the SG that have a value of 1.
2. *Quantitative Solving* means computing the values of all vertices in the SG.
3. *Strategic Solving* means computing the optimal strategy for Eve or Adam.

Graph Representation of Stochastic Games: All graphs of stochastic games follow a consistent notation (see Figure 2 for examples):



(a) Example SPG graph. Vertices with priority 0 are blue, ones with priority 1 are red.



(b) Example SSG graph. Target vertices are yellow, non-target vertices are white.

Figure 2: Example SGs.

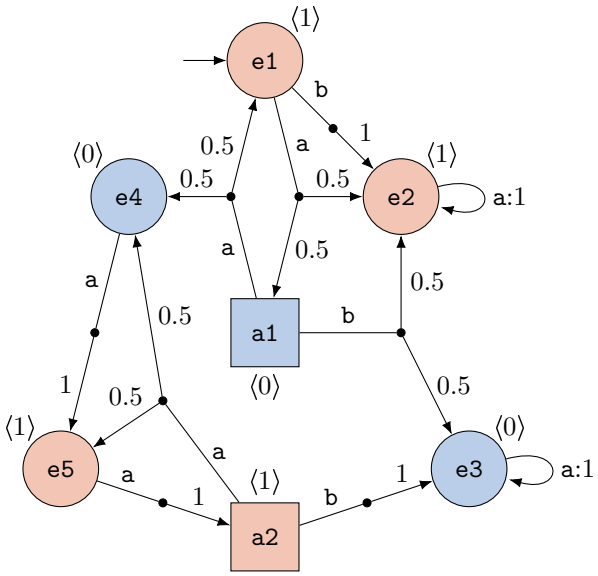


Figure 3: SPG graph. Vertices with priority 0 are illustrated in blue, ones with priority 1 in red.

- Eve vertices are represented by circles, and Adam vertices by squares.
- The initial vertex is indicated by an arrow pointing to it from no preceding node.
- Transitions are split into two parts: the first part is labeled with the action, and the second part illustrates the probability distribution over the successor vertices.
  – Deterministic transitions are shown as a single arrow from the first part to a successor vertex, labeled with probability 1.
  – Probabilistic transitions are shown as multiple arrows from the first part to different successor vertices, each labeled with its corresponding probability.
  For a self-loop with action a, this is simplified with one arrow labeled with $a : 1$.

In SPGs, the priority $p(v)$ of a vertex $v \in V$ is indicated using the label $\langle p(v) \rangle$.
In SSGs, target vertices are marked with the label $\langle \mathrm{T} \rangle$, while non-target vertices remain unlabeled. Occasionally, we use colors to enhance visual clarity, e.g., to highlight priorities in SPGs or target vertices in SSGs. The specific color scheme varies between figures and is explained in the respective captions.

Example of Stochastic Parity Game: We discuss the example in Figure 3 that occurs several times in the following sections. Therefore, we intuitively describe how SPGs work.

Assume that Eve tries to win with even parity. Let us first consider `e4`, `e5`, and `a2`.
Independent of Adam's choice in `a2`, Eve wins with probability 1. If Adam chooses `b`, the game moves to `e3` where it loops infinitely often with priority 0. If Adam chooses `a`, the game will visit `e4` infinitely often with probability 1, thus the smallest priority that is seen is even. Therefore, if we reach `e4`, `e5`, or `a2`, Eve wins with probability 1.

The game starts in `e1`. Eve does not choose action `b`, since this way she loses because we infinitely often loop on priority 1 in `e2`. Therefore, Eve chooses action `a`. With probability 0.5, we reach `e2` and Eve loses anyway. With probability 0.5, we reach `a1`.
In `a1`, if Adam chooses action `b`, Eve loses with probability 0.5 and wins with probability 0.5 since we reach `e3`. Adam does not choose action `a` in `a1` because Eve directly wins with probability 0.5 by reaching `e4` as shown before and with the remaining probability of 0.5, we reach `e1` again where the play started. The probability of winning from `e1` is higher than 0 when Adam always chooses action `a` in `a1` and therefore it is better for Adam to choose action b in `a1`.
This results in a winning probability of 0.25 for Eve when trying to achieve even parity.

4 Specification of SPGs and SSGs

In Stargate, the user inputs the SPG or SSG via a specification file. This file contains all necessary information for creating an SPG or SSG. For this purpose we create specification languages for SPGs and SSGs in this section.
For the basic use case of Stargate, a specification language for SSGs is not required, as there is no need to save an SSG when solving an SPG directly. Nonetheless, we introduce such a language to support intermediate storage of SSGs and allow continuing the transformation process at a later stage.

4.1 SPG Specification

In Stargate, an SPG has three attributes. The first is a set of vertices each having a unique name. A label indicates if it is an Eve or Adam vertex. Each vertex's priority is part of the vertex object itself for easier handling. A vertex is uniquely identified with its name. Second, there is a set of transitions each consisting of a starting vertex, an action and a set of probabilities and corresponding end vertices. When the probabilities do not sum up to one, the specification is invalid. The probability can be expressed in decimal or common fraction notation. Each transition is uniquely identified through its starting vertex and its action. Lastly, we specify the initial vertex.

For the specification, we define the **SP**G specification **lang**uage *SPLANG* with a grammar. This grammar defines the overall structure of SPLANG. However, Stargate accepts specifications that have additional whitespace, e.g., at the start of a line, between parameters or empty lines in between.

We define the following symbols:

Non-Terminals :	SPLANG, EV, AV, VD, IV, TS, TD, ENV
Terminals :	`a`, ..., `z`, `A`, ..., `Z`, `0`, ..., `9`, [**s**], [**pt**], [**pb**], ␣, `:`, `+`, `|`, \n
Start Symbol :	SPLANG

The symbols have the following semantics:

- SPLANG: represents the whole SPG specification in SPLANG
- EV: represents the vertex declaration for player Eve
- AV: represents the vertex declaration for player Adam
- VD: represents one vertex declaration (recursive)
- IV: represents the declaration of the initial vertex
- TS: represents the SPG's transition declaration
- TD: represents one transition declaration (recursive)
- ENV: represents the end vertices of one transition (recursive)
- [**s**]: is a string of arbitrary characters (except space), used to represent vertex names and actions
- [**pt**]: is a non-negative integer representing a vertex's priority
- [**pb**]: is an evaluable float expression in $\mathbb{Q}_{\geq 0}$
- ␣: represents a whitespace character
- \n: represents a line break

In the production rules, we enclose sequences of terminals with ‘ ’, insert space around non-terminals, and visualize line breaks for better readability and comprehension.
The rules are as follows:

Grammar 4.1.1: SPLANG

SPLANG → ‘spg\n’ EV ‘\n’ AV ‘\n’ IV ‘\n’ TS

EV → ‘evevertices\n’ VD ‘endevevertices’

AV → ‘adamvertices\n’ VD ‘endadamvertices’

VD → ‘[s] : [pt]\n’ VD | ε

IV → ‘initialvertex : [s]’

TS → ‘transitions\n’ TD ‘endtransitions’

TD → ‘[s] [s] : [s]\n’ TD | ‘[s] [s] : [pb] | [s]’ ENV ‘\n’ TD | ε

ENV → ‘ + [pb] | [s]’ ENV | ε

We will now look at the structure of a SPLANG specification step by step.

Vertex Declaration: The SPG specification file name has to end with ".spg" and the file has to start with `spg`, similar to PRISM SMG specification files. The keyword `evevertices` is in a separate line. What follows are the vertices of the SPG where Eve chooses the next action. Each vertex is declared in a separate line.
The names of the vertices must not be separated by whitespaces, since STARGATE parses the lines with space as separators between the parameters. All vertex names also have to be different in order to offer unique identification of the vertices. After every vertex name, we have a colon followed by a priority. The value of this priority is a non-negative integer. To end the vertex declaration for Eve, the keyword `endevevertices` is in a separate line after the last declared Eve vertex. For the case of an SPG where Adam has control over all vertices and Eve has none, the content of the Eve vertices section can be left empty but the keywords `evevertices` and `endevevertices` have to occur in the file nevertheless.

As described above, the declaration of Eve vertices follows this pattern:

```
evevertices
<vertexname> : <prio>
...
endevevertices
```

What follows is the keyword `adamvertices` after which the vertices for Adam are declared, analogous to the declaration of the vertices of Eve. The vertices' names have to be different from each other and from the vertices' names of Eve. The section is then ended with `endadamvertices`. Likewise, this section can be empty but the keywords have to occur.

Analogously, the declaration of Adam vertices follows this pattern:

```
adamvertices
<vertexname> : <prio>
...
endadamvertices
```

Initial Vertex Declaration: After that, the initial vertex is declared with "`initialvertex :` " and the vertex name which has to be one of the previously declared ones.

Therefore, the initial vertex declaration follows the pattern:

```
initialvertex : <vertexname>
```

Transition Declaration: Lastly, the declaration of transitions begins with `transitions` and ends with `endtransitions`. In between, each transition is declared in a separate line. A declaration can be written in two ways. The first is a simplified format for deterministic transitions which means that the set of end vertices contains only one vertex with probability 1. It begins with the starting vertex's name, followed by the action, then a colon and the end vertex's name. The second format is for probabilistic transitions with multiple end vertices. The transition begins with the starting vertex's name followed by the action and a colon. After this, the set of end vertices is described where each combination of probability and end vertex is separated by a "+". One combination is written with the probability first, then "|" and, last, the end vertex. Since the transitions are uniquely identified, they must differ in starting vertex, action or both.

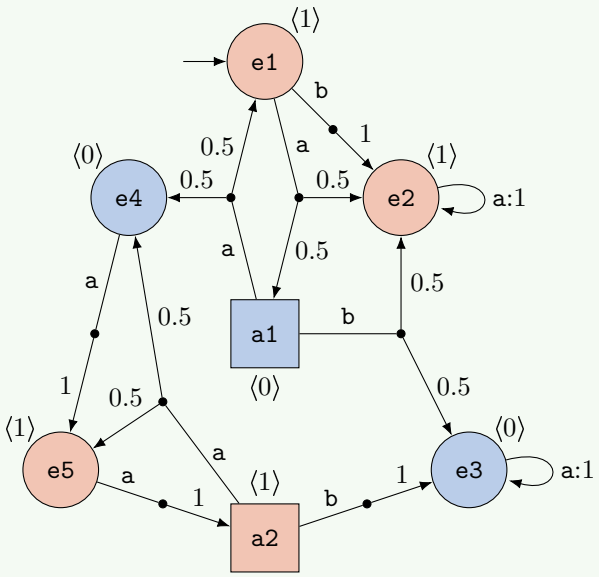
The declaration of transitions is according to the following pattern:

```
transitions
<vertexname> <action> : <vertexname>
<vertexname> <action> : <prob> | <vertexname> + ... + <prob> | <vertexname>
...
endtransitions
```

This results in a specification that can be read by Stargate.
Example 4.1.1 illustrates an SPG specification in SPLANG and the corresponding graph.

Example 4.1.1: SPG specification in SPLANG (left) **and corresponding graph** (right)

```
spg

evevertices
e1 : 1
e2 : 1
e3 : 0
e4 : 0
e5 : 1
endevevertices

adamvertices
a1 : 0
a2 : 1
endadamvertices

initialvertex : e1

transitions
e1 a : 0.5 | a1 + 0.5 | e2
e1 b : e2
e2 a : e2
a1 a : 0.5 | e1 + 0.5 | e4
a1 b : 0.5 | e3 + 0.5 | e2
e4 a : e5
e5 a : a2
a2 a : e3
e3 a : e3
a2 b : 0.5 | e5 + 0.5 | e4
endtransitions
```



(Vertices with priority 0 are illustrated in blue, with priority 1 in red.)

4.2 SSG Specification

In addition to SPLANG, we define a specification language for SSGs. This allows Stargate to transform SPGs into SSGs and save them for later use. The user can then load the SSG specification and continue the solving process.
An SSG is very similar to an SPG in Stargate. Thus, the **SS**G specification **lang**uage *SSLANG* is very similar to SPLANG (see Grammar 4.1.1), too. The only difference lies in the treatment of vertices: instead of assigning priorities, each vertex is labeled to indicate whether it is a target. This is done in the `evevertices` or `adamvertices` section by appending a "T" (case-insensitive) after the name of the vertex declaration.

We define SSLANG similar to SPLANG. The differences are:

- In the terminals, instead of **[pt]** we have **[T]**, being either ε or 'T'. This represents the target attribute for a vertex.
- In the production rules, we redefine VD the following way:

Grammar 4.2.1: SSLANG

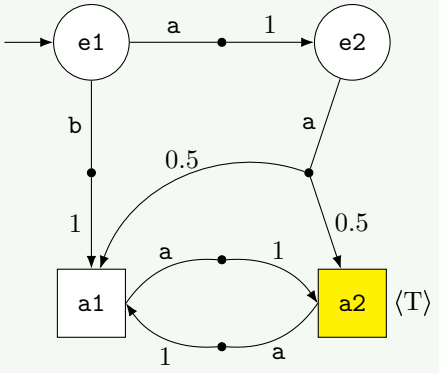
$$
\begin{aligned}
&\dots \\
&\text{VD} \rightarrow \text{'[s]}\textvisiblespace\text{[T]}\backslash\text{n'} \\
&\qquad\quad \text{VD} \\
&\qquad\quad \mid \varepsilon \\
&\dots
\end{aligned}
$$

(see Grammar 4.1.1 for production rules of other symbols)

Just as for SPLANG, STARGATE accepts specifications that have additional whitespace.
An example SSG specified in SSLANG with the respective graph is illustrated in Example 4.2.1

Example 4.2.1: SSLANG Vertex Declaration

```
ssg
evevertices
e1
e2
endevevertices
adamvertices
a1
a2 T
endadamvertices
initialvertex : e1
transitions
e1 a : e2
e1 b : a1
e2 a : 0.5 | a1 + 0.5 | a2
a1 a : a2
a2 a : a1
endtransitions
```



(Non-target vertices are illustrated in white, target vertices in yellow.)

Methods for Specifications: STARGATE offers several methods for SPG and SSG specifications:

- reading an SPG/SSG specification from a specification file and creating the respective SPG/SSG object
- converting an SPG/SSG object into an SPG/SSG specification
- saving an SPG/SSG specification into an SPG/SSG specification file
- reformatting an existing SPG/SSG specification with respect to indentation and spacing

Creation of SPGs and SSGs: For the direct creation of a game, STARGATE performs several sanity checks. The user can set a flag that checks for the creation process of an SSG if it contains vertices for Eve and Adam, respectively. This is necessary for the solving process to be correct. When following the reduction process correctly, this check should never fail. Additionally, for every creation of a transition for either of the game types, STARGATE makes sure that the respective probabilities sum up to 1 and that the probabilities are positive. Furthermore, STARGATE can convert floats to exact values with fractions. This is kept through the whole process if wanted up until the solving process. Lastly, it is checked if the SSG has a deadlock vertex, i.e., a vertex that has no outgoing transitions. This is important for the resulting SMG since PRISM-games does not allow deadlock states.

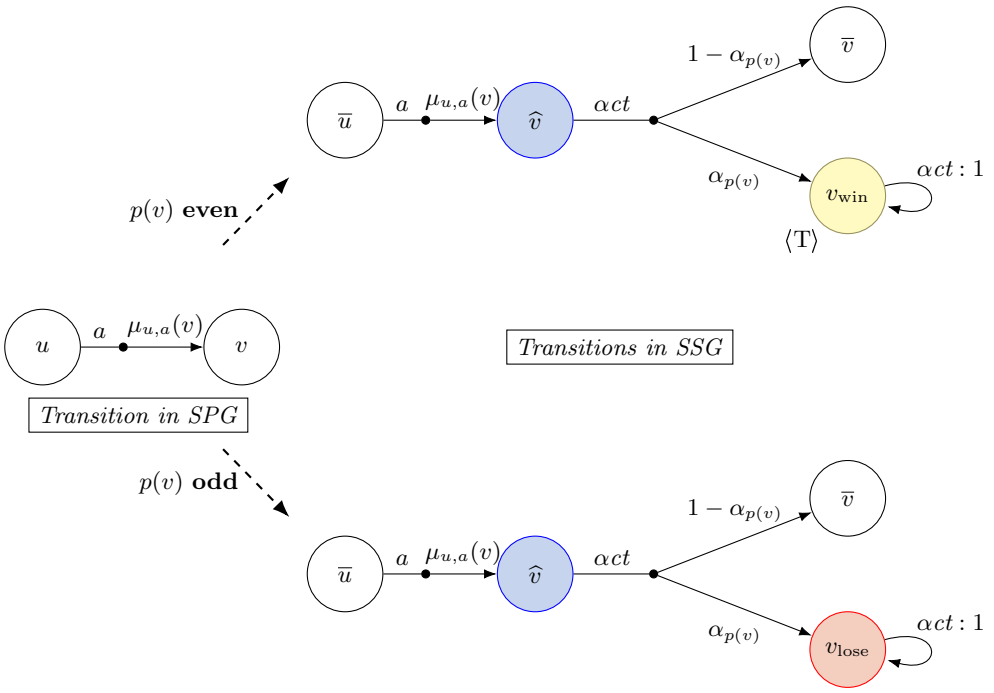


Figure 4: Gadget replacing a transition with action $a \in Av(u)$ and probability $\mu_{u,a}(v)$ to go to v with a based on the parity of $p(v)$.

5 Reduction from SPGs to SSGs

After specifying the SPG, we perform the reduction from the SPG $(G, PA(p))$ into the resulting SSG $(\widetilde{G}, RE(\{v_{win}\}))$. Note that for simplification we occasionally call the SPG G and the resulting SSG $\widetilde{G}$. The following section is based on the reduction presented by Berthon et al. [3].

Overview: The goal of the reduction is the transformation of an SPG into an SSG such that the optimal strategy for reaching a new target vertex in the SSG coincides with an optimal strategy in the SPG. To obtain this reductive behavior, we use a gadget that ensures the correctness. This gadget construction still requires the computation of an *alpha function* $\alpha : \text{Prio}((G, PA(p))) \to \mathbb{Q}$ where $\text{Prio}((G = (V, \ldots), PA(p)))$ denotes $\{n \in \mathbb{N} \mid n \text{ occurs as a priority value } p(v) \text{ for a } v \in V\}$. This function is based on the arena of the SPG and the priority values. Occasionally, we represent the values of this function with $\alpha_0, \ldots, \alpha_{|\text{Prio}((G,PA(p)))|-1}$ which are the respective *alpha values* for the priorities in the SPG from smallest to largest. Note that not always $\alpha_0 = \alpha(0)$ since the smallest priority does not have to be 0.
Ultimately, the reduction consists of two steps:

1. Compute the alpha function based on the SPG.
2. Transform the SPG into an SSG using the gadget construction.

We start by discussing the theoretical background of the gadget construction and the alpha function, then look at the implementation in Stargate.

5.1 Theoretical Background

Gadget Transformation: The gadget seen in Figure 4 was presented first by Chatterjee et al. [5]. Let V be a set of vertices of a stochastic arena. Intuitively, we duplicate every vertex $v \in V$ of the SPG into $\widehat{v}$ and $\overline{v}$. For $U \subseteq V$ we denote $\overline{U} = \{\overline{v} \mid v \in U\}$ and $\widehat{U} = \{\widehat{v} \mid v \in U\}$. We call $\widetilde{V}$ the set of transformed vertices, $\overline{V}$ the set of duplicate vertices, and $\widehat{V}$ the set of intermediate vertices.
Every path that visits v in the SPG first visits $\widehat{v}$ in the SSG and with a high probability $\overline{v}$ afterwards. With a low probability the play may instead reach one of the *game deciding vertices* v_{win} or v_{lose}. Which one is visited depends on $p(v)$. v_{win} has a chance to be visited if $p(v)$ is even and v_{lose} has a chance to be visited if $p(v)$ is odd. Both are sinks and the game can be seen as finished if one of them is reached. Thus, we check the winning event of the SPG by checking the reaching event of v_{win}.

Finishing a play by reaching v_{win} represents a win for the player that tries to achieve even parity. Without loss of generality, we assume that Eve tries to achieve even parity. Reaching v_{lose} represents a win for the player with the odd parity objective.
Intuitively, this means that the player is rewarded a small chance of winning when the priority they target is seen in the play. This probability of ending the game is dependent on the alpha values which we will discuss later in this section.
Formally, we define the resulting SSG as follows:

Definition 5.1.1: Resulting SSG $(\widetilde{G}, \widetilde{\Phi})$ from Gadget Transformation

Let (G, Φ) be an SPG where $G = (V, V_\exists, V_\forall, v_I, A, Av, \delta)$ is a stochastic arena and $\Phi = PA(p), p : V \to \mathbb{N}$ is a priority function on the arena's vertices.

We construct the SSG $(\widetilde{G} = (\widetilde{V}, \widetilde{V}_\exists, \widetilde{V}_\forall, \widetilde{v}_I, \widetilde{A}, \widetilde{Av}, \widetilde{\delta}), \widetilde{\Phi})$ as follows:

$$
\begin{aligned}
\widetilde{V} &= \widetilde{V}_\exists \uplus \widetilde{V}_\forall \\
\widetilde{V}_\exists &= \overline{V}_\exists \uplus \widehat{V}_\forall \uplus \{v_{win}\} \\
\widetilde{V}_\forall &= \overline{V}_\forall \uplus \widehat{V}_\exists \uplus \{v_{lose}\} \\
\widetilde{v}_I &= \overline{v}_I \\
\widetilde{A} &= A \uplus \alpha ct \\
\widetilde{Av} &: \widetilde{V} \to 2^{\widetilde{A}}, \widetilde{Av}(\widetilde{v}) \mapsto \begin{cases} Av(v) & \text{if } \widetilde{v} \in \overline{V} \\ \{\alpha ct\} & \text{otherwise} \end{cases} \\
\widetilde{\delta} &: \widetilde{V} \times \widetilde{A} \to \mathscr{D}(\widetilde{V}), (\widetilde{u} \in \widetilde{V}, a \in \widetilde{Av}(\widetilde{u})) \mapsto \widetilde{\mu}_{\widetilde{u},a}, \\
\text{where } \widetilde{\mu}_{\widetilde{u},a}(\widetilde{v}) &= \begin{cases} \mu_{u,a}(v) & \text{if } \widetilde{u} \in \overline{V}, \widetilde{v} \in \widehat{V} \\ 1 - \alpha_{p(u)} & \text{if } \widetilde{u} \in \widehat{V}, \widetilde{v} \in \overline{V}, u = v \\ \alpha_{p(u)} & \text{if } \widetilde{u} \in \widehat{V}, \widetilde{v} = v_{win}, p(u) \text{ is even} \\ \alpha_{p(u)} & \text{if } \widetilde{u} \in \widehat{V}, \widetilde{v} = v_{lose}, p(u) \text{ is odd} \\ 1 & \text{if } \widetilde{u} = \widetilde{v} = v_{win} \text{ or } \widetilde{u} = \widetilde{v} = v_{lose} \end{cases} \\
\widetilde{\Phi} &= RE(\{v_{win}\})
\end{aligned}
$$

We determine the alpha function later in Lemma 11 from [3] and Lemma 5.1.2. We discuss Definition 5.1.1 for clarification:

- $\widetilde{V}$ is the disjoint union of $\widetilde{V}_\exists$ and $\widetilde{V}_\forall$ following Definition 3.2.1 for stochastic arenas. Therefore, we now give further details on $\widetilde{V}_\exists$ and $\widetilde{V}_\forall$.
- $\widetilde{V}_\exists$ is composed of $\overline{V}_\exists$, $\widehat{V}_\forall$, and the vertex v_{win}. $\overline{V}_\exists$ has to be in $\widetilde{V}_\exists$, because these are duplicate vertices that can have multiple actions that Eve can choose from. $\widehat{V}_\forall$ is in $\widetilde{V}_\exists$ because all vertices in this set have only one available action, namely αct and therefore they can be assigned to any player. We choose Eve as this benefits the transformation process (see Section 6.2.1). The same reasoning applies for v_{win} as we benefit from v_{win} to be controlled by Eve and therefore to be in $\widetilde{V}_\exists$.
  An analogous construction applies to $\widetilde{V}_\forall$ and Adam.
- $\widetilde{v}_I$ is the duplicate vertex of the initial vertex v_I in the stochastic arena G. Notice that we start at the duplicate $\overline{v}_I$ and not at the intermediate vertex $\widehat{v}_I$.
- $\widetilde{A}$ extends A from G with the action αct. This action is used for every additional transition that appears in the gadget construction. This is also inferred from the definition of $\widetilde{Av}$. For every duplicate vertex $\overline{v} \in \overline{V}$, with $\overline{v}$ is the duplicate vertex of $v \in V$, we assign the actions $Av(v)$ of v. For any other vertex we always have exactly one outgoing transition with action αct.
- $\widetilde{\delta}$ is defined by five different cases depending on the starting vertex and the end vertex or end vertices. We consider the different cases in the definition of $\widetilde{\mu}_{\widetilde{u},a}(\widetilde{v})$, which is defined the following way:

  1. $\mu_{u,a}(v)$ if the start vertex is a duplicate and the end vertex is an intermediate vertex. This value is given by the stochastic arena G.

2. $1-\alpha_{p(u)}$ if the start vertex is an intermediate vertex and the end vertex is the respective duplicate vertex. a is αct in this case.

3./4. $\alpha_{p(u)}$ if the start vertex is an intermediate vertex and the end vertex is either v_{win} or v_{lose} depending on the parity value $p(u)$.

5. 1 if the transition is a self-loop on the game deciding vertices.

- $\widetilde{\Phi}$ is the reachability objective for Eve for reaching the winning vertex v_{win}.

In the following section we consider a definition for the alpha function ensuring the reduction to be correct.

Computing the alpha values: For the computation of the alpha values, we have to build up more terms and theorems. Therefore, the following section discusses the work of Berthon et al. [3]. Let SPG $\mathcal{G} = (G, PA(p))$ and SSG $\widetilde{\mathcal{G}} = (\widetilde{G}, RE(\{v_{win}\}))$ and $\sigma \in \Sigma_\exists, \gamma \in \Sigma_\forall$ be strategies in $\mathcal{G}$ for Eve and Adam, respectively. For shorter notation, we write $\mathbb{P}_{\sigma,\gamma}$ for the probability of Eve to win $\mathcal{G}$ with the even parity winning objective, and $\widetilde{\mathbb{P}}_{\sigma,\gamma}$ for the probability of Eve to win $\widetilde{\mathcal{G}}$ by reaching v_{win}.

We use the definitions of BSCCs, even BSCCs and odd BSCCs from Section 2.1 in [3]:

Definition 5.1.2: Bottom Strongly Connected Components (BSCCs) (from Section 2.1 in [3])

Let MC $M = (V, \delta, v_I)$.

A set $S \subseteq V$ is *strongly connected* if for all pairs of states $v, v' \in S$, v and v' are mutually reachable.

A set $S \subseteq V$ is a *strongly connected component (SCC)* if it is strongly connected and there exists no other set $L \subseteq V$ with $S \subsetneq L$ and L is strongly connected.

A set $S \subseteq V$ is a *bottom SCC (BSCC)* if S is an SCC and there does not exist $v \in S, v' \in V \setminus S$ such that $\delta(v, v') > 0$.

Let $M = (V, \delta, v_I)$ be the induced MC from applying strategies σ, γ for Eve and Adam respectively to an SPG $(G, PA(p))$.

A set $S \subseteq V$ is an *even BSCC* if S is a BSCC in M and $\min(p(S))$ is even. *Odd BSCCs* are defined analogously.

The paper [3] introduces the expressions $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath)$, $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven)$, and $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd)$ where

- $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath)$ denotes the probability for a play starting from $\overline{v} \in \overline{V}$ to reach a BSCC in $\widetilde{G}$,
- $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven)$ denotes the minimum probability of reaching the winning sink after reaching the respective even BSCC, and
- $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd)$ denotes the maximum probability of reaching the winning sink after reaching the respective odd BSCC.

In Lemma 7 they introduce a lower bound for $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven)$ and an upper bound for $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd)$. These bounds help prove Lemma 8 which shows for fixed strategies $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$ the relation between the winning probability $\mathbb{P}_{\sigma,\gamma}$ in G and the winning probability $\widetilde{\mathbb{P}}_{\sigma,\gamma}$ in $\widetilde{G}$ depending on $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath)$, $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven)$, and $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd)$:

▶ **Lemma 8** from [3]

If there exists $x, y \in (0,1)$ such that $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > x$, $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) \geq y$ for all even k, and $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd) \leq 1 - y$ for all odd k, it holds:

$$y \cdot \mathbb{P}_{\sigma,\gamma} - y + x \cdot y \leq \widetilde{\mathbb{P}}_{\sigma,\gamma} \leq \mathbb{P}_{\sigma,\gamma} + 1 - x \cdot y$$

We split this lemma up and create our own:

Lemma 5.1.1

Let $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$ be strategies for players Eve and Adam, respectively. If there exists $x, y \in (0,1)$ such that $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > x$, $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) \geq y$ for all even k, and $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd) \leq 1 - y$ for all odd k, it holds:

$$\widetilde{\mathbb{P}}_{\sigma,\gamma} \geq y \cdot \mathbb{P}_{\sigma,\gamma} - y + x \cdot y \quad (a)$$
$$\widetilde{\mathbb{P}}_{\sigma,\gamma} \leq \mathbb{P}_{\sigma,\gamma} + 1 - x \cdot y \quad (b)$$

Afterwards, Lemma 9 proves that for $(\sigma, \gamma), (\sigma', \gamma') \in \Sigma_\exists \times \Sigma_\forall$ if the respective winning probabilities $\mathbb{P}_{\sigma,\gamma}$ and $\mathbb{P}_{\sigma',\gamma}$ differ, they differ by at least $\frac{1}{(n!)^2 M^{2n^2}}$, where

- n denotes the number of vertices $|V|$ in G, and
- M denotes the maximum denominator $b_{\mu_{v,a}(u)}$, when all transitional probabilities $\mu_{v,a}(u)$ are represented as a rational number $\mu_{v,a}(u) = \frac{a_{\mu_{v,a}(u)}}{b_{\mu_{v,a}(u)}}$.

The paper continues with Theorem 10. This theorem acts as the center of the reduction:

▶ **Theorem 10** from [3]
If for all $(\sigma, \gamma) \in \Sigma_\exists \times \Sigma_\forall$, and $v \in V$ the following conditions hold:

1. $\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > \frac{4-\varepsilon}{4}$
2. $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) \geq \frac{4}{4+\varepsilon}$ for all even k, and $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd) \leq 1 - \frac{4}{4+\varepsilon}$ for all odd k

where $\varepsilon = \frac{1}{(n!)^2 M^{2n^2}}$, then every optimal strategy $\sigma \in \Sigma_\exists$ of Eve in the SSG $(\widetilde{G}, RE(\{v_{win}\}))$ is also optimal in the SPG $(G, PA(p))$. The same holds for Adam.

We see that this theorem only concerns the reduction for the optimal strategy.
For optimal strategies, ε becomes extraordinarily small with increasing vertex space. This led to problems during the implementation. Therefore we introduce an extension of Theorem 10 from [3] for ε-optimality.
A strategy $\sigma \in \Sigma_\exists$ is called *ε-optimal* for Eve if the respective winning probability $\mathbb{P}_{\sigma,\gamma^*}$ of this strategy differs by less than ε from the winning probability $\mathbb{P}_{\sigma^*,\gamma^*}$ for an optimal strategy $\sigma^* \in \Sigma_\exists$, i.e., if $|\mathbb{P}_{\sigma,\gamma^*} - \mathbb{P}_{\sigma^*,\gamma^*}| < \varepsilon$, where $\gamma^* \in \Sigma_\forall$ is an optimal strategy for Adam. The analogous holds for Adam.

Theorem 5.1.1: Reducing SPGs to SSGs with ε-Optimality

If for all $(\sigma, \gamma) \in \Sigma_\exists \times \Sigma_\forall$ the following conditions hold:

(1) $\widetilde{\Pr}_{\sigma,\gamma}(crossPath) > \frac{4-2\varepsilon}{4-\varepsilon}$

(2) $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) \geq \frac{4-\varepsilon}{4}$ for all even k, and $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winOdd) \leq 1 - \frac{4-\varepsilon}{4}$ for all odd k

where $\varepsilon \in (0,1)$, then every optimal strategy $\sigma \in \Sigma_\exists$ of Eve in the SSG $(\widetilde{G}, RE(\{v_{win}\}))$ is ε-optimal in the SPG $(G, PA(p))$. The same holds for Adam.

Proof. We assume Conditions (1) and (2) hold. This gives the values for $x = \frac{4-2\varepsilon}{4-\varepsilon}$ and $y = \frac{4-\varepsilon}{4}$.
We show that every optimal strategy $\sigma \in \Sigma_\exists$ in $\widetilde{G}$ is ε-optimal in G. We prove this by contraposition. This means that we show that if a strategy is not ε-optimal in G, then it is not optimal in $\widetilde{G}$.
Let $(\sigma, \gamma) \in \Sigma^*_\exists \times \Sigma^*_\forall$ where $\Sigma^*_\exists \subseteq \Sigma_\exists$ and $\Sigma^*_\forall \subseteq \Sigma_\forall$ denotes the sets of optimal strategies for Eve and Adam, respectively. Let $(\sigma', \gamma') \in \Sigma_\exists \times \Sigma_\forall$ and $\varepsilon > 0$.
We prove that if $\mathbb{P}_{\sigma,\gamma} - \mathbb{P}_{\sigma',\gamma'} > \varepsilon$ (3) then $\widetilde{\mathbb{P}}_{\sigma,\gamma} - \widetilde{\mathbb{P}}_{\sigma',\gamma'} > 0$, i.e., if $\mathbb{P}_{\sigma',\gamma'}$ is not ε-optimal in G then $\widetilde{\mathbb{P}}_{\sigma',\gamma'}$ is not optimal in $\widetilde{G}$.

From Lemma 5.1.1 (a) and (b) we infer

$$\begin{aligned} \widetilde{\mathbb{P}}_{\sigma,\gamma} &\geq y \cdot \mathbb{P}_{\sigma,\gamma} - y + x \cdot y && (4) \\ \text{and} \quad \widetilde{\mathbb{P}}_{\sigma',\gamma'} &\leq \mathbb{P}_{\sigma',\gamma'} + 1 - x \cdot y && (5) \end{aligned}$$

Therefore

$$\begin{aligned} \widetilde{\mathbb{P}}_{\sigma,\gamma} - \widetilde{\mathbb{P}}_{\sigma',\gamma'} \geq\ & y \cdot \mathbb{P}_{\sigma,\gamma} - y + x \cdot y - \mathbb{P}_{\sigma',\gamma'} - 1 + x \cdot y && \text{by (4) and (5)} \\ =\ & \frac{4-\varepsilon}{4} \cdot \mathbb{P}_{\sigma,\gamma} - \frac{4-\varepsilon}{4} + \frac{4-2\cdot\varepsilon}{4-\varepsilon} \cdot \frac{4-\varepsilon}{4} - \mathbb{P}_{\sigma',\gamma'} - 1 \\ & + \frac{4-2\cdot\varepsilon}{4-\varepsilon} \cdot \frac{4-\varepsilon}{4} \\ =\ & \frac{4-\varepsilon}{4} \cdot \mathbb{P}_{\sigma,\gamma} - \frac{4-\varepsilon}{4} + 2 \cdot \frac{4-2\cdot\varepsilon}{4} - \mathbb{P}_{\sigma',\gamma'} - 1 + \mathbb{P}_{\sigma,\gamma} - \mathbb{P}_{\sigma,\gamma} && \text{by (1) and (2)} \\ >\ & \frac{(4-\varepsilon)\cdot\mathbb{P}_{\sigma,\gamma} - 4 + \varepsilon + 8 - 4\varepsilon - 4 - 4\cdot\mathbb{P}_{\sigma,\gamma}}{4} + \varepsilon && \text{by (3)} \\ =\ & \frac{-\mathbb{P}_{\sigma,\gamma}\cdot\varepsilon}{4} - \frac{3\cdot\varepsilon}{4} + \varepsilon \\ =\ & \frac{\varepsilon}{4} - \frac{\mathbb{P}_{\sigma,\gamma}\cdot\varepsilon}{4} \\ =\ & \frac{\varepsilon}{4} \cdot (1 - \mathbb{P}_{\sigma,\gamma}) \\ >\ & 0 \end{aligned}$$

We proved that under conditions (1) and (2) every optimal strategy $\sigma \in \Sigma_\exists$ in $\widetilde{G}$ is ε-optimal in G. □

After Theorem 10 in [3], the paper discusses in Lemma 11 how the alpha function is restricted in order to ensure that Conditions 1 and 2 from Theorem 10 are satisfied. We define δ_{min} as the smallest occurring probability greater than zero on a transition in the arena, formally $\delta_{min} = \min(\{\mu_{v,a}(u) \mid u, v \in V, a \in Av(v), u \in e(v,a)\}$.

▶ **Lemma 11** from [3]
When the values of α are arranged as follows, the conditions in Theorem 10 are satisfied:

1. If $\alpha_0 \leq \frac{\delta_{min}^n}{8(n!)^2 M^{2n^2}}$, then Condition 1 is satisfied.
2. If for all $k \in \mathbb{N}$, the following holds, then Condition 2 is satisfied:
   $\frac{\alpha_{k+1}}{\alpha_k} \leq \frac{\delta_{min}^n(1-\delta_{min})}{8(n!)^2 M^{2n^2}+1}$

This gives us a bound for the alpha values if we solve the SPG for optimal strategies. Since we defined an extension of Theorem 10 of [3] for ε-optimal strategies, we need to find respective bounds for alpha values such that the conditions from Theorem 5.1.1 are satisfied. For this we need Corollary 23 and Corollary 24 from [3]:

▶ **Corollary 23** from [3]
For all strategy pairs $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, for all $\overline{v} \in \overline{V}$, the following holds:

$$\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > \frac{\delta_{min}^n(1-\alpha_0)^{n+1}}{2\alpha_0 + \delta_{min}^n(1-\alpha_0)^{n+1}}$$

▶ **Corollary 24** from [3]
For all strategy pairs $\sigma, \gamma \in \Sigma_\exists \times \Sigma_\forall$, for all even k, the following holds:

$$\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) > \frac{\delta_{min}^n(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta_{min}^n(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}}$$

Now we introduce our own alpha values for ε-optimality:

Lemma 5.1.2: Alpha Values for ε-Optimality

When the values of α are arranged as follows, the conditions in Theorem 5.1.1 are satisfied:

I. If $\alpha_0 \leq \dfrac{4 \cdot \varepsilon \cdot \delta_{min}^n}{(4-\varepsilon) \cdot 8}$ then Condition 1 is satisfied.

II. If for all $k \in \mathbb{N}$ the following holds, then Condition 2 is satisfied:
$\dfrac{\alpha_{k+1}}{\alpha_k} \leq \dfrac{\delta_{min}^n \cdot (1-\delta_{min})}{\frac{8\cdot(4-\varepsilon)}{4\cdot\varepsilon} + 1}$

Proof. We first prove I.

By Corollary 23 from [3], we know:

$$\widetilde{\Pr}^{\overline{v}}_{\sigma,\gamma}(crossPath) > \frac{\delta_{min}^n (1-\alpha_0)^{n+1}}{2\alpha_0 + \delta_{min}^n (1-\alpha_0)^{n+1}}.$$

In order to show Condition 1 of Theorem 5.1.1, which is

$$\widetilde{\Pr}_{\sigma,\gamma}(crossPath) > \frac{4-2\varepsilon}{4-\varepsilon}$$

it suffices to show

$$\frac{\delta_{min}^n (1-\alpha_0)^{n+1}}{2\alpha_0 + \delta_{min}^n (1-\alpha_0)^{n+1}} > \frac{4-2\varepsilon}{4-\varepsilon}.$$

We rearrange this the following way:

$$\begin{aligned}
& \frac{\delta_{min}^n (1-\alpha_0)^{n+1}}{2 \cdot \alpha_0 + \delta_{min}^n (1-\alpha_0)^{n+1}} > \frac{4-2\cdot\varepsilon}{4-\varepsilon} = \frac{4-\frac{4\cdot\varepsilon}{4-\varepsilon}}{4} \\
\iff & \frac{4 \cdot \delta_{min}^n \cdot (1-\alpha_0)^{n+1}}{2 \cdot \alpha_0 + \delta_{min}^n \cdot (1-\alpha_0)^{n+1}} - 4 > -\frac{4\cdot\varepsilon}{4-\varepsilon} \\
\iff & \frac{4\cdot\varepsilon}{4-\varepsilon} > \frac{8\cdot\alpha_0 + 4\cdot\delta_{min}^n\cdot(1-\alpha_0)^{n+1} - 4\cdot\delta_{min}^n(1-\alpha_0)^{n+1}}{2\cdot\alpha_0 + \delta_{min}^n\cdot(1-\alpha_0)^{n+1}} \\
\iff & \frac{4\cdot\varepsilon}{4-\varepsilon} > \frac{8\cdot\alpha_0}{2\cdot\alpha_0 + \delta_{min}^n\cdot(1-\alpha_0)^{n+1}}.
\end{aligned}$$

Now we show that if $\alpha_0 \leq \frac{4\cdot\varepsilon\cdot\delta_{min}^n}{(4-\varepsilon)\cdot 8}$ holds, then $\frac{4\cdot\varepsilon}{4-\varepsilon} > \frac{8\cdot\alpha_0}{2\cdot\alpha_0+\delta_{min}^n\cdot(1-\alpha_0)^{n+1}}$ holds as well:

$$\begin{aligned}
& \frac{8\cdot\alpha_0}{2\cdot\alpha_0 + \delta_{min}^n\cdot(1-\alpha_0)^{n+1}} \\
\leq\ & \frac{8\cdot\alpha_0}{2\cdot\alpha_0 + \delta_{min}^n\cdot(1-(n+1)\cdot\alpha_0)} && \text{(Bernoulli's inequality)} \\
=\ & \frac{8\cdot\alpha_0}{\alpha_0 + \delta_{min}^n + \alpha_0\cdot(1-(n+1)\cdot\delta_{min}^n)} \\
\leq\ & \frac{8\cdot\alpha_0}{\alpha_0 + \delta_{min}^n} && (1-(n+1)\delta_{min}^n > 0) \\
\leq\ & \frac{8\cdot\frac{4\cdot\varepsilon\cdot\delta_{min}^n}{(4-\varepsilon)\cdot 8}}{\frac{4\cdot\varepsilon\cdot\delta_{min}^n}{(4-\varepsilon)\cdot 8} + \delta_{min}^n} && \left(\alpha_0 \leq \tfrac{4\cdot\varepsilon\cdot\delta_{min}^n}{(4-\varepsilon)\cdot 8}\right) \\
=\ & \frac{1}{\frac{1}{8} + \delta_{min}^n\cdot\frac{4-\varepsilon}{4\cdot\varepsilon\cdot\delta_{min}^n}} && \left(\tfrac{1}{8} > 0\right) \\
<\ & \frac{1}{\frac{4-\varepsilon}{4\cdot\varepsilon}} \\
=\ & \frac{4\cdot\varepsilon}{4-\varepsilon}
\end{aligned}$$

From this follows I.

Now we prove II. By Corollary 24 from [3] we know:

$$\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) > \frac{\delta^n_{min}\cdot(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}}.$$

In order to show Condition 2 of Theorem 5.1.1, which is

$$\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) \geq \frac{4-\varepsilon}{4}$$

we show

$$\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) > \frac{4-\varepsilon}{4}$$

which implies Condition 2. For this it suffices to show:

$$\frac{\delta^n_{min}\cdot(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}\cdot(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} \geq \frac{4-\varepsilon}{4}$$

We rearrange this the following way:

$$\begin{aligned}
& \frac{\delta^n_{min}(1-\delta_{min}) - \frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} && \geq && \frac{4-\varepsilon}{4} = \frac{4}{4+\frac{4\cdot\varepsilon}{4-\varepsilon}} \\
\iff & \frac{(\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}) - 2\cdot\frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} && \geq && \frac{(4+\frac{4\cdot\varepsilon}{4-\varepsilon}) - \frac{4\cdot\varepsilon}{4-\varepsilon}}{4+\frac{4\cdot\varepsilon}{4-\varepsilon}} \\
\iff & \frac{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} - \frac{2\cdot\frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} && \geq && \frac{4+\frac{4\cdot\varepsilon}{4-\varepsilon}}{4+\frac{4\cdot\varepsilon}{4-\varepsilon}} - \frac{\frac{4\cdot\varepsilon}{4-\varepsilon}}{4+\frac{4\cdot\varepsilon}{4-\varepsilon}} \\
\iff & 1 - \frac{2\cdot\frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} && \geq && 1 - \frac{\frac{4\cdot\varepsilon}{4-\varepsilon}}{4+\frac{4\cdot\varepsilon}{4-\varepsilon}} \\
\iff & \frac{\frac{4\cdot\varepsilon}{4-\varepsilon}}{4+\frac{4\cdot\varepsilon}{4-\varepsilon}} && \geq && \frac{2\cdot\frac{\alpha_{k+1}}{\alpha_k}}{\delta^n_{min}(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}} \\
\iff & \frac{4\cdot\varepsilon}{4-\varepsilon}\cdot\left(\delta^n_{min}\cdot(1-\delta_{min}) + \frac{\alpha_{k+1}}{\alpha_k}\right) && \geq && 2\cdot\frac{\alpha_{k+1}}{\alpha_k}\cdot\left(4+\frac{4\cdot\varepsilon}{4-\varepsilon}\right) \\
\iff & \frac{4\cdot\varepsilon}{4-\varepsilon}\cdot\left(\delta^n_{min}\cdot(1-\delta_{min})\right) && \geq && 8\cdot\frac{\alpha_{k+1}}{\alpha_k} + \frac{\alpha_{k+1}}{\alpha_k}\cdot\frac{4\cdot\varepsilon}{4-\varepsilon} \\
\iff & \frac{\alpha_{k+1}}{\alpha_k} && \leq && \frac{\frac{4\cdot\varepsilon}{4-\varepsilon}\cdot\delta^n_{min}\cdot(1-\delta_{min})}{8+\frac{4\cdot\varepsilon}{4-\varepsilon}} \\
\iff & \frac{\alpha_{k+1}}{\alpha_k} && \leq && \frac{\delta^n_{min}\cdot(1-\delta_{min})}{\frac{8\cdot(4-\varepsilon)}{4\cdot\varepsilon}+1}
\end{aligned}$$

From this follows that $\frac{\alpha_{k+1}}{\alpha_k} \leq \frac{\delta^n_{min}\cdot(1-\delta_{min})}{\frac{8\cdot(4-\varepsilon)}{4\cdot\varepsilon}+1}$ implies $\widetilde{\Pr}^{k}_{\sigma,\gamma}(winEven) \geq \frac{4-\varepsilon}{4}$, which proves II. □

This proves the achievability of our alpha values. This reduction is only suitable for **stochastic** parity games meaning at least one transition must have a probability strictly less than one. The numerator for the computation of the alpha values $\frac{\alpha_{k+1}}{\alpha_k}$ is a product with $1-\delta_{min}$ for both optimal and ε-optimal strategies. If $\delta_{min} = 1$ then $(1-\delta_{min} = 0)$, thus $\frac{\alpha_{k+1}}{\alpha_k} = 0$. As a result, all alpha values that are not for the minimum priority occurring become zero which leads to the reduction to become incorrect since only the respective game deciding vertex for the smallest priority can be reached. If a game has no transition with a probability smaller than one, then the game has to be solved using a solver for specifically deterministic parity games, e.g., PGSOLVER [13].

5.2 Reduction from SPGs to SSGs in Stargate

For the transformation, Stargate takes two main arguments: the input SPG $(G, PA(p))$, and ε for the ε-optimality. If ε is not assigned a specific value then Stargate performs the reduction with the alpha values from the original paper [3].
The algorithm for the transformation uses a subalgorithm to compute the alpha values, which we consider first. The reduction algorithm passes down the arguments $(G, PA(p))$ and ε to the α computing algorithm. Algorithm 5.2.1 follows a simple structure where first δ_{min} is defined and after that the alpha values are computed following Lemma 11 from [3] and Lemma 5.1.2.
In the algorithm, we assume the following:

Algorithm 5.2.1 Compute alpha values

Input: SPG $(G = (V, V_\exists, V_\forall, v_I, A, Av, \delta), \Phi = PA(p))$, ε (optional)
Output: $\alpha : \text{Prio}((G, \Phi)) \to \mathbb{Q}$

1: $\delta_{min} :=$ *minimum probability occurring in* G
2: **if** ε value is not defined **then**
3: # For optimal strategy
4: $M :=$ *maximum denominator as described in Section 5.1*
5: $\alpha_0 := \frac{\delta_{min}^n}{8 \cdot (n!)^2 \cdot M^{2 \cdot n^2}}$
6: **for all** k in $\{0, \ldots, q-1\}$ **do**
7: $\alpha_{k+1} := \frac{\delta_{min}^n \cdot (1-\delta_{min})}{8 \cdot (n!)^2 \cdot M^{2 \cdot n^2} + 1} \cdot \alpha_k$
8: **end for**
9: **else**
10: # For ε-optimal strategy
11: $\alpha_0 := \frac{4 \cdot \varepsilon \cdot \delta_{min}^n}{(4-\varepsilon) \cdot 8}$
12: **for all** k in $\{0, \ldots, q-1\}$ **do**
13: $\alpha_{k+1} := \frac{\delta_{min}^n \cdot (1-\delta_{min})}{\frac{8 \cdot (4-\varepsilon)}{4 \cdot \varepsilon} + 1} \cdot \alpha_k$
14: **end for**
15: **end if**
16: **return** α

- Let $\{p_0, \ldots, p_q\} = \text{Prio}((G, PA(p)))$ be the set of occurring priorities in ascending order where with $q = |\text{Prio}((G, PA(p)))| - 1$. When we define $\alpha_k := x$ this means we set $\alpha(p_k) := x$. This way we define the function α by defining the alpha values.
- The definition of δ_{min} is abstracted here. In STARGATE, a different function returns the minimum probability. The same holds for M.

We continue with the algorithm that performs the actual reduction: We assume the following for actions in Algorithm 5.2.2:

- For every update of $\widetilde{V}_\exists$ and $\widetilde{V}_\forall$, $\widetilde{V}$ is updated automatically such that $\widetilde{V} := \widetilde{V}_\exists \cup \widetilde{V}_\forall$. The same holds for $\overline{V}$ and $\widehat{V}$.
- For every update of $\widetilde{\delta}$, $\mathcal{T}(\widetilde{G})$ is updated automatically with $\mathcal{T}(\widetilde{G}) := \{t = (v, a, \mu) \mid v \in \widetilde{V}, a \in \widetilde{Av}(v), \widetilde{\delta}(v, a) = \mu\}$ and $\widetilde{Av}(v)$ is updated for every $v \in \widetilde{V}$ with $\widetilde{Av}(v) := \{a \mid \exists t \in \mathcal{T}(\widetilde{G}) \text{ with } t = (v, a, \mu)\}$.
- For every update of $\widetilde{Av}(v)$, $\widetilde{A}$ is updated with $\widetilde{A} = \bigcup_{\widetilde{v} \in \widetilde{V}} \widetilde{Av}(\widetilde{v})$.
- For every Eve vertex $v \in V_\exists$ it holds that $\overline{v} \in \overline{V}_\exists$ and $\widehat{v} \in \widehat{V}_\forall$. Analogously, for Adam vertices $v \in V_\forall$ it holds that $\overline{v} \in \overline{V}_\forall$ and $\widehat{v} \in \widehat{V}_\exists$.

We further clarify Algorithm 5.2.2:

- In the for-loop starting in Line 2, we define for each vertex $v \in V$ the respective duplicate vertex $\overline{v} \in \overline{V}$ and intermediate vertex $\widehat{v} \in \widehat{V}$.
- DV is short for *duplicate vertex*. This function maps a vertex in G to its duplicate vertex in $\widetilde{G}$.
- IV is short for *intermediate vertex*. This function maps the duplicate vertex $\overline{v}$ of v to the intermediate vertex $\widehat{v}$ of v. Note that when concatenating DV and IV, for a vertex v in G we get the intermediate vertex $\widehat{v}$ of v.
- In Line 8, we define the initial vertex $\widetilde{v}_I$ of $\widetilde{G}$, which is the duplicate vertex of v_I in G.
- In Lines 10 to 13, we define the game deciding vertices v_{win} and v_{lose} and their self-loops.
- In the for-loop starting in Line 14, we add the transitions that originate from G. If there is a transition in G from v to u then $\widetilde{G}$ has a transition from $DV(v)$ to $IV(DV(u))$.
- Finally, in the for-loop starting in Line 21, we define the transitions between a vertex's intermediate and duplicate vertex. When the priority $p(v)$ of the vertex v is even, then the transition has v_{win} as one of its end vertices with respective probability $\alpha(p(v))$. If $p(v)$ is odd, then one of the end vertices is v_{lose} with respective probability $\alpha(p(v))$. For both cases we reach the respective duplicate vertex with probability of $1 - \alpha(p(v))$.

Algorithm 5.2.2 SPG to SSG Reduction

Input: SPG $(G = (V, V_\exists, V_\forall, v_I, A, Av, \delta), PA(p))$, ε optional
Output: SSG $(\widetilde{G} = (\widetilde{V}, \widetilde{V}_\exists, \widetilde{V}_\forall, \widetilde{v}_I, \widetilde{A}, \widetilde{Av}, \widetilde{\delta}), RE(\{v_{win}\})$

Compute alpha function following Algorithm 5.2.1
for all v in V **do**
    $\overline{V} := \overline{V} \cup \{\overline{v}\}$ with new vertex $\overline{v}$
    $DV(v) := \overline{v}$
    $\widehat{V} := \widehat{V} \cup \{\widehat{v}\}$ with new vertex $\widehat{v}$
    $IV(\overline{v}) := \widehat{v}$
end for
$\widetilde{v}_I := DV(v_I)$
$\widetilde{V} := \overline{V} \cup \widehat{V}$
$\widetilde{V}_\exists := \widetilde{V}_\exists \cup \{v_{win}\}$ with new vertex v_{win}
$\widetilde{\delta}(v_{win}, \alpha ct) := \mu_{win} : \widetilde{V} \to \mathbb{Q}_{\geq 0}, \mu_{win}(v_{win}) = 1$
$\widetilde{V}_\forall := \widetilde{V}_\forall \cup \{v_{lose}\}$ with new vertex v_{lose}
$\widetilde{\delta}(v_{lose}, \alpha ct) := \mu_{lose} : \widetilde{V} \to \mathbb{Q}_{\geq 0}, \mu_{lose}(v_{lose}) = 1$
for all $t = (v, a, \mu_{v,a})$ in $\mathcal{T}(G)$ **do**
    Initialize $\mu_{new} : \widehat{V} \to \mathbb{Q}_{\geq 0}$
    for all v_e in $e(t)$ **do**
        $\mu_{new}(IV(DV(v_e))) := \mu_{v,a}(v_e)$
    end for
    $\widetilde{\delta}(DV(v), a) := \mu_{new}$
end for
for all v in V **do**
    Initialize $\mu_{new} : \widetilde{V} \to \mathbb{Q}_{\geq 0}$
    $\mu_{new}(DV(v)) := 1 - \alpha(p(v))$
    if $p(v)$ is even **then**
        $\mu_{new}(v_{win}) := \alpha(p(v))$
    else
        $\mu_{new}(v_{lose}) := \alpha(p(v))$
    end if
    $\widetilde{\delta}(IV(DV(v)), \alpha ct) := \mu_{new}$
end for

After performing our reduction, we obtain the SSG. The next step is to solve the SSG in order to reason about the input SPG.

6 PRISM-games Integration

This section discusses the integration of PRISM-games [21] into STARGATE. First in Section 6.1, we talk about the version of PRISM-games STARGATE uses, then in Section 6.2 we present three approaches that transform SSGs into *stochastic multiplayer games (SMGs)* which is a PRISM-games-readable format. Finally in Section 6.3, we describe the methods in STARGATE that use PRISM-games to analyze SMGs.

6.1 Solving with PRISM-games

For solving the SSGs, we decided for the tool PRISM-games [21]. This tool is based on and extends the probabilistic model checker PRISM [20]. When solving an SMG with PRISM-games, the user can choose the used solving algorithm [21]. The only implemented solving algorithm in the current version of PRISM-games is *Value Iteration*. As shown by Haddad and Monmege [15], Value Iteration has trouble converging on certain instances for both an absolute and relative convergence criterion. During the development of STARGATE, we found that this imprecision is amplified for the very small alpha values in the SMGs we create. Thus, Value Iteration is not the optimal solving method for this purpose.
Therefore, we consider the PRISM-games extension by Křetínský et al. [18]. This extension offers additional solving algorithms. These are *Gauss-Seidel Value Iteration* [19], *Policy Iteration*, *Modified Policy Iteration* [24], *Interval Iteration*, *Sound Value Iteration*, and *Topological Value Iteration*.
In the current version of PRISM-games (Version 3.2.1), there is an option to export the computed strategy when solving the SMG. This option is not available in the extension presented by Křetínský et al. [18] since it is based on an earlier version of PRISM-games.

6.2 Transformation from SSGs to SMGs

We transform an SSG that resulted from our reduction into an SMG which are one of several models that PRISM-games uses [21].

Definition 6.2.1: Stochastic Multiplayer Games

An SMG is characterized through a tuple $\mathcal{G} = (\Pi, S, (S_i)_{i\in\Pi}, \overline{s}, A, \delta, L)$ where

- Π is a finite set of players,
- S is a finite set of states,
- $(S_i)_{i\in\Pi}$ is a partition with respect to the players of S,
- $\overline{s} \in S$ is an initial state,
- A is a finite set of actions,
- $\delta : S \times A \to Dist(S)$ is a (partial) probabilistic transition function, and
- $L : S \to 2^{AP}$ is a labeling function mapping states to sets of atomic propositions from a defined set AP.

We observe that this definition has several similarities to the definition of SSGs. Both models have states or vertices, initial states or vertices and a transition relation that is defined with probability distributions. PRISM models are defined textually, utilizing guarded command notation [21]. When we use the term vertex we refer to the vertices of an SSG and state when we refer to the state space of an SMG.
The model has more than one state in the sense of a status that is given by the value of the set of variables of the module. These variables change according to the guarded commands in the format:

```
[act] guard -> p_1 : update_1 + ... p_n : update_n;
```

with $\sum_{i=1}^{n} p_i = 1$ and each update being a collection of assignments of the module's variables. For instance, consider a robot that moves on a two-dimensional grid. Its current position is given by two variables x and y. Every move it has a chance of 0.9 to successfully move to the next square.

For the remaining 0.1 it stays on the current square. In guarded command notation this behavior is represented as:

```
[moveup] (x=0 & y=0) -> 0.9 : (y'=1) | 0.1 : true;
```

In this context, `true` means that no variable is updated. One major difference between SSGs as we defined them in Section 3.2 and SMGs is synchronization. Using PRISM-games, we observed that one iteration of a play in an SMG works as follows:

1. Determine which player is the one to choose the action. This is done by checking for all guarded commands whether the current state satisfies the guard. The available actions result from the available guarded commands.
2. If two or more players can perform an action specified in their own respective module, PRISM-games raises an error. Nevertheless, an action from player 1 can be the action of a guarded command in a module controlled by player 2. This is due to synchronization.
3. When the action is chosen, first the update of the module of the player that had control over the current state is performed. After that, all other players update their own variables according to the guarded command.

A variable x from module m can only be modified in this module m. Since every module is controlled by exactly one player, every variable can only be modified by that player.
An intuitive approach for transforming SSGs to SMGs is to keep track of the vertices with a variable in a module. We show that this does not work:
We assume (without loss of generality) that we have one variable in Eve's module. In this case, Eve can update the state in accordance with the guarded commands and the actions available to her. Through these updates, the state variable, i.e., the current vertex, can be modified.
Now suppose Eve performs an action such that the play reaches a vertex controlled by Adam. Now it is Adam's turn to perform an action according to his guarded commands and actions. In order for the play to transition to the next vertex, Adam needs to have the ability to modify the state variable.
This is not the case because the variable is in Eve's module. Thus, the SSG cannot be easily modeled as an SMG in PRISM-games with only one variable for the state.
In the process of finding a correct and simple transformation, we came up with three different versions. The first two are very similar. The third one takes a different approach. We now look at all three of them by examining their semi-formal pseudo code algorithms.

6.2.1 Transformation Version 1: Alternating Vertices (improved)

The first version is based on alternating vertices.

Definition 6.2.2: Alternating Transitions

A transition $t = (v, a, \mu)$ where $v \in V_\exists$ is *alternating* if

$$\nexists v' \in V_\exists \text{ such that } v' \in e(t)$$

and when $v \in V_\forall$

$$\nexists v' \in V_\forall \text{ such that } v' \in e(t).$$

or t is a self-loop.
A stochastic arena G is alternating if for all $t \in \mathcal{T}(G)$ holds that t is alternating. An SG (G, Φ) is alternating if the respective arena G is alternating.

Intuitively, alternating means that in a play there is a constant alternation between vertices that are controlled by Eve and vertices that are controlled by Adam.
In the following we will go through the transformation process from an SSG to an SMG with the first version. The transformation is split into two phases. First, we modify the SSG to be alternating and second, we perform the actual transformation from the alternating SSG to the SMG.

Algorithm 6.2.1 Version 1, Phase 1: Modifying SSG to be alternating (1)

Input: SSG object $(G = (V, V_\exists, V_\forall, v_I, A, Av, \delta), RE(T))$
Output: SSG object $(G' = (V', V'_\exists, V'_\forall, v'_I, A', Av', \delta'), RE(T'))$ which is alternating

```
1:  G' := G
2:  T' := T
3:  A' := A' ⊎ {extra_eve_action}
4:  A' := A' ⊎ {extra_adam_action}
5:  seen := ∅
6:  for all t = (v, a, μ) ∈ T(G') do
7:      for all v_e in e(t) do
8:          if v_e ∈ seen then
9:              continue
10:         else if v = v_e and |e(t)| = 1 then
11:             continue
12:         else if v ∈ V_∃ and v_e ∈ V_∃ then
13:             seen := seen ∪ {v_e}
14:             v_new be a new vertex, V'_∀ := V'_∀ ∪ {v_new}
15:             IV(v_e) := v_new
16:         else if v ∈ V_∀ and v_e ∈ V_∀ then
17:             seen := seen ∪ {v_e}
18:             v_new be a new vertex, V'_∃ := V'_∃ ∪ {v_new}
19:             IV(v_e) := v_new
20:         end if
21:     end for
22: end for
```

Version 1, Phase 1: We have a look at the first phase. Since the algorithm is quite long, we split it into two parts.
We assume that the following actions are performed automatically in both parts:

- For every update of $V_\exists$ and $V_\forall$, V is updated automatically with $V := V_\exists \cup V_\forall$
- For every update of δ, the set of transitions in the arena $\mathcal{T}(G)$ is updated automatically with $\mathcal{T}(G) := \{t = (v, a, \mu) \mid v \in V, a \in Av(v), \delta(v, a) = \mu\}$. This means that a transition $t = (v, a, \mu)$ can simply be replaced by redefining the value of $\delta(v, a)$.
- For every update of transitions, $Av(v)$ is updated for every $v \in V$ with $Av(v) := \{a \mid \exists t \in \mathcal{T}(G) \text{ with } t = (v, a, \mu)\}$.
- For every update of $Av(v)$, A is updated with $A = \bigcup_{v \in V} Av(v)$.
- In a for-loop, the set that is iterated over is not affected by updates inside this for-loop. The updates are cached and applied after the for-loop.

In Algorithm 6.2.1, we perform the following steps:

1. We introduce fresh actions, namely extra_eve_action and extra_adam_action, to ensure that for the modifications the actions are not used before in the SSG. This enables the possibility to later trace back the actions in the SMG to the original actions in the SSG. Both players need an additional action since SMGs allow actions only to be owned by one player.
2. For each vertex, we check whether there is an incoming transition that is not alternating. Note that we ignore self-loops as we defined them to be alternating in Definition 6.2.2. When there is a non-alternating transition, we add an intermediate vertex that is controlled by the respective other player. In order to do this effectively, in *seen* we cache vertices that we already now will receive an intermediate vertex such that we continue if we encounter these.

We perform the following steps in Algorithm 6.2.2:

For each transition, we check if this transition has to be replaced. If the transition is deterministic, this is done by comparing the controlling players of the starting vertex and the only end vertex, respectively in Lines 3-11 . Two transitions are added and the old one is removed. The first one is from the starting vertex of the original transition to the intermediate vertex with the action and the probability of the original action.
The second new transition starts at the intermediate vertex and goes to the end vertex with one

Algorithm 6.2.2 Version 1, Phase 1: Modifying SSG to be alternating (2)

for all $t = (v, a, \mu) \in \mathcal{T}(G')$ **do**
  if t is deterministic **then**
    if t is a self-loop **then**
      continue
    else if $v \in V'_\exists$ and $v_{t,1} \in V'_\exists$ **then**
      $\delta'(v, a) := \mu_{new}$ where $\mu_{new}(IV(v_{t,1})) = 1$
      $\delta'(IV(v_{t,1}), \text{extra\_adam\_action}) := \mu_{new}$ where $\mu_{new}(v_{t,1}) = 1$
    else if $v \in V'_\forall$ and $v_{t,1} \in V'_\forall$ **then**
      $\delta'(v, a) := \mu_{new}$ where $\mu_{new}(IV(v_{t,1})) = 1$
      $\delta'(IV(v_{t,1}), \text{extra\_eve\_action}) := \mu_{new}$ where $\mu_{new}(v_{t,1}) = 1$
    end if
  else
    if $v \in V'_\exists$ **then**
      Initialize $\mu_{end} : e(t) \to \mathbb{Q}_{\geq 0}$
      for all $v_e \in e(t)$ **do**
        if $v_e \in V'_\exists$ **then**
          $\delta'(IV(v_e), \text{extra\_adam\_action}) := \mu_{new}$ where $\mu_{new}(v_e) = 1$
          $\mu_{end}(IV(v_e)) := \mu(v_e)$
        else
          $\mu_{end}(v_e) := \mu(v_e)$
        end if
      end for
      $\delta'(v, a) := \mu_{end}$
    else
      Initialize $\mu_{end} : e(t) \to \mathbb{Q}_{\geq 0}$
      for all $v_e \in e(t)$ **do**
        if $v_e \in V'_\forall$ **then**
          $\delta'(IV(v_e), \text{extra\_eve\_action}) := \mu_{new}$ where $\mu_{new}(v_e) = 1$
          $\mu_{end}(IV(v_e)) := \mu(v_e)$
        else
          $\mu_{end}(v_e) := \mu(v_e)$
        end if
      end for
      $\delta'(v, a) := \mu_{end}$
    end if
  end if
end for
return $(G', RE(T'))$

of the extra actions. After that, the original transition is removed from the SSG.
If the transition is a self-loop we continue with the next transition.
In case that the transition is probabilistic, the program iterates over the end vertices and similarly collects the new end vertices. For every end vertex we have two possibilities:

i. If the end vertex v is controlled by the same player as the starting vertex, we store the intermediate vertex $IV(v)$.

ii. If the end vertex v is controlled by a different player than the player controlling the starting vertex, we store the end vertex v.

After iterating over all end vertices we create a new transition that replaces the old one and has the stored vertices as end vertices with their respective probability. While deterministic transitions could be handled the same, we treat them separately to reduce the number of operations.
From an SSG with n vertices and m transitions, the worst case is that the SSG is completely controlled by one player and every transition is a probabilistic transition with all n vertices in the end vertex set. For every vertex, one intermediate vertex and one transition are added. For every original transition, one transition is added while one is removed. That means from the SSG with n vertices and m transition we obtain an SSG with $n+n\cdot 1 = 2n$ vertices and $m+n+m\cdot(1-1) = m+n$ transitions.

Version 1, Phase 2: After transforming the original SSG to be alternating, we create the respective SMG specification. We split this algorithm into several parts, since it is very long.

Algorithm 6.2.3 Version 1, Phase 2: Create SMG specification (1)

Input: Alternating SSG object $(G = (V, V_\exists, V_\forall, v_I, A, Av, \delta), RE(T))$
Output: SMG specification

```
1:  statecount_∃ := 1
2:  statecount_∀ := 1
3:  states_∃ := ∅
4:  states_∀ := ∅
5:  for all v ∈ V do
6:      if v ∈ V_∃ then
7:          states_∃ := states_∃ ∪ {(statecount_∃, 0)}
8:          LV(v) := statecount_∃
9:          RV(v) := 0
10:         statecount_∃ := statecount_∃ + 1
11:     else
12:         states_∀ := states_∀ ∪ {(0, statecount_∀)}
13:         LV(v) := 0
14:         RV(v) := statecount_∀
15:         statecount_∀ := statecount_∀ + 1
16:     end if
17: end for
```

For all parts we assume the following:

- For strings s, t, we denote the concatenation of these with $s + t$. In this context, $s \mathrel{+}= t$ denotes $s := s + t$. If t is a suffix of s then s - t denotes s' where $s' + t = s$. Analogously, $s \mathrel{-}= t$ means $s := s$ - t.
- Our representation is similar to f-strings in Python. A string is embraced by “”. If we have evaluable expressions inside we embrace these inside the f-string with $\langle\rangle$. Note that if “” is in a different font, e.g., `""`, compared to the ”” that signal the f-string, then these `""` are characters in the f-string.
- “\n” denotes a line break.

We discuss the performed steps in Algorithm 6.2.3 for further clarification:

1. In Lines 1-4, we define counters for the states and sets for the states for Eve and Adam respectively.
2. In the for-loop beginning in Line 5, we transform each vertex into its state representation for the SMG. We achieve this by enumerating all vertices of Eve and Adam, respectively. Then, we map the enumerated vertices of Eve to tuples of the format $(n, 0)$ where $n \in \{1, \ldots, |V_\exists|\}$. Similarly, we map Adam vertices in the format $(0, m)$.
   Additionally, we save for each mapped state the values of both entries of the tuples. $LV(v)$ (*left value*) is the left entry of the tuple for the respective state of v and $RV(v)$ (*right value*) is the right entry.

We explain further explain Algorithm 6.2.4. The performed steps are similar to the first part, but instead of vertices, we transform actions:

1. In Lines 1-6, we define counters for the actions and sets where to collect the actions for Eve and Adam, respectively.
2. In the for-loop beginning in Line 7, we search through all transitions which action is used by which player and collect them. It is possible that an action is used by both players. We handle this in the next step.
3. In the for-loops beginning in Lines 14 and 19, we transform the actions used by Eve and Adam, respectively. The actions that Eve uses, are renamed to `e`n where $n \in \{1, \ldots, |ssgacts_\exists|\}$. We perform a similar process for Adam with actions `a`m, $m \in \{1, \ldots, |ssgacts_\forall|\}$.

In Algorithm 6.2.5 we start creating the SMG specification. We perform the following steps:

1. In Line 1, we start by adding the keyword `smg` in the first line. PRISM-games requires this to distinguish between different game formats.

Algorithm 6.2.4 Version 1, Phase 2: Create SMG specification (2)

```
1: actcount_∃ := 1
2: actcount_∀ := 1
3: ssgacts_∃ := ∅
4: ssgacts_∀ := ∅
5: smgacts_∃ := ∅
6: smgacts_∀ := ∅
7: for all t = (v, a, μ) ∈ T(G) do
8:     if v ∈ V_∃ then
9:         ssgacts_∃ := ssgacts_∃ ∪ {a}
10:    else
11:        ssgacts_∀ := ssgacts_∀ ∪ {a}
12:    end if
13: end for
14: for all a ∈ ssgacts_∃ do
15:    smgacts_∃ := smgacts_∃ ∪ {"e⟨actcount_∃⟩"}
16:    act_∃(a) := "e⟨actcount_∃⟩"
17:    actcount_∃ := actcount_∃ + 1
18: end for
19: for all a ∈ ssgacts_∀ do
20:    smgacts_∀ := smgacts_∀ ∪ {"a⟨actcount_∃⟩"}
21:    act_∀(a) := "a⟨actcount_∃⟩"
22:    actcount_∀ := actcount_∀ + 1
23: end for
```

Algorithm 6.2.5 Version 1, Phase 2: Create SMG specification (3)

```
1: spec := "smg\n"
2: spec += "player eve\n"
3: spec += "evemod, [e1], ...,[e⟨actcount_∃ − 1⟩]\n"
4: spec += "endplayer\n"
5: spec += "player adam\n"
6: spec += "adammod, [a1], ..., [a⟨actcount_∀ − 1⟩]\n"
7: spec += "endplayer\n"
```

2. In Lines 2-4, we add the player Eve. In this player declaration, we assign the transformed actions and the module that is defined in the next part.

3. In Lines 5-7, we analogously add player Adam.

In Algorithm 6.2.6, we create the modules. This comprises the transformation of the transitions. For simplification, in the transition transformation we only examine the case if the starting vertex of the transition is controlled by Eve, since the case for Adam is very similar.
We perform the following steps:

1. In Lines 1 and 2, we begin defining the module for Eve, namely `evemod`, which contains the variable declaration for `es`. `es` is the variable that represents at which Eve vertex the play currently is. If the play is at an Adam vertex, then `es` is 0 and `as` in `adammod` is greater than zero. A similar process holds for `adammod`.
   Both `es` and `as` are initialized according to the respective state of the initial vertex v_I in G.

2. In the for-loop from Line 5, we iterate over all transitions $\mathcal{T}(G)$ in G.
   Without loss of generality we assume that the transition we examine is controlled by Eve. The case for Adam is analogous.

   2.1. If the start vertex is deterministic, we check whether the transition is a self-loop in Line 8.

   If yes, we add guarded commands to both modules that represent the self-loop from Lines 9-11. This is done by keeping the values for `es` and `as`.
   The reason why we defined self-loops to be alternating earlier in this section is this part. In all other cases we transfer the responsibility for the next action to Adam, but here is no need for this. Afterwards, we continue with transforming the next transition.

Algorithm 6.2.6 Version 1, Phase 2: Create SMG specification (4)

```
1: emod = "module evemod\n"
2: emod += "es :  [0..⟨max(1, statecount_∃ − 1)⟩] init ⟨LV(v_I)⟩;\n"
3: amod = "module adammod\n"
4: amod += "as :  [0..⟨max(1, statecount_∀ − 1)⟩] init ⟨RV(v_I)⟩;\n"
5: for all t = (v, a, μ) ∈ T(G) do
6:     if v ∈ V_∃ then
7:         if |e(t)| = 1 then
8:             if e(t) = {v} then
9:                 amod += "[⟨act_∃(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> true;\n"
10:                emod += "[⟨act_∃(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> true;\n"
11:                continue
12:            else
13:                amod += "[⟨act_∃(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> (as'=⟨RV(v_{t,1})⟩);\n"
14:            end if
15:        else
16:            amod += "[⟨act_∃(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> "
17:            for all v_e ∈ e(t) do
18:                amod += "(⟨μ(v_e)⟩) :  (as'=⟨RV(v_e)⟩) + "
19:            end for
20:            amod = (amod - " + ") + "\n"
21:        end if
22:        emod += "[⟨act_∃(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> (es'=0);\n"
23:    else
24:        if |e(t)| = 1 then
25:            if e(t) = {v} then
26:                emod += "[⟨act_∀(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> true;\n"
27:                amod += "[⟨act_∀(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> true;\n"
28:                continue
29:            else
30:                emod += "[⟨act_∀(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> (es'=⟨LV(v_{t,1})⟩);\n"
31:            end if
32:        else
33:            emod += "[⟨act_∀(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> "
34:            for all v_e ∈ e(t) do
35:                emod += "(⟨μ(v_e)⟩) :  (es'=⟨LV(v_e)⟩) + "
36:            end for
37:            emod = (emod - " + ") + "\n"
38:        end if
39:        amod += "[⟨act_∀(a)⟩] (es=⟨LV(v)⟩ & as=⟨RV(v)⟩) -> (as'=0);\n"
40:    end if
41: end for
42: spec += emod + "endmodule\n" + amod + "endmodule\n"
```

If no, we add a guarded command to `adammod` that lets Adam change his variable to the respective RV of the next state.

2.2. If the start vertex is not deterministic, we collect the probabilities for each end vertex and add them to the guarded command (Lines 16-20). The probabilistic action is performed by Adam since he is the one to change the variables.
Note that this does not empower Adam to choose the next vertex since this is still done by Eve.

At the end of the transformation in Line 22, we add a guarded command in `evemod` that means that for every transition from an Eve vertex, Eve sets her own state variable to 0 to signal that she passes the responsibility for the next action to Adam.

3. We concatenate all modules with the specification in Line 42.

In Algorithm 6.2.7 we collect the target vertices. In SMGs we can assign attributes to states with labels. Therefore, we add the label `"target"` that is true for one state if both values (LV and

Algorithm 6.2.7 Version 1, Phase 2: Create SMG specification (5)

1: spec += "`label "target" = (`"
2: **for all** $v \in T$ **do**
3: spec += "`(es=`$\langle LV(v) \rangle$`) & (as=`$\langle RV(v) \rangle$`) | `"
4: **end for**
5: **if** exists string s' with s' + "`(`" = spec **then**
6: spec = (spec - "`(`") + "`false;`"
7: **else**
8: spec = (spec - "` | `") + "`);`"
9: **end if**
10: **return** spec

RV) match the values of one of the target vertices.
Note that when doing the reduction of an SPG to an SMG, we always have one target vertex which is v_{win}. This target vertex label is used later for solving the SMG which translates to solving the respective SPG.

6.2.2 Transformation Version 2: Alternating Vertices (initial)

The second version of transforming an SSG into an SMG originates from the first idea of using alternating vertices. This means that Version 2 is very similar to Version 1.
In fact, the only difference from Version 1 is the transformation from the SSG to its alternating representation. In Version 2, instead of adding an intermediate vertex for each vertex, we repeat this for every transition if needed. Therefore, for the second phase of the transformation, we refer to Algorithms 6.2.3 to 6.2.7.
We make the same assumptions as for Version 1 declared in Section 6.2.1. These assumptions define the behavior for automatically updating the vertex set V, the set of transitions $\mathcal{T}(G)$, the parameters concerning actions A and Av, and the behavior of iterators in for-loops.
Now, we redefine the first phase for Version 2: The difference to Algorithm 6.2.8 from Version 1 is highlighted.
In Version 1 (Algorithm 6.2.1) we determine in which vertices we need an intermediate vertex, i.e., there is an incoming non-alternating transition. Afterwards, we create this vertex and reference it with help of the function IV. After that, in Algorithm 6.2.2 we use this referenced vertex $IV(v)$ for vertex v to transform the transition such that they are alternating. In Version 2, we create a separate vertex for each transition.
This way, Version 1 never adds more vertices or transitions. Nevertheless, since Version 2 does not filter for vertices v that need an intermediate vertex $IV(v)$ in the first place, Version 2 is shorter and saves some overhead from omitting this filtering process. The key difference between Version 1 and 2 is that in Version 1 we add one intermediate for every vertex if necessary and in Version 2 we add one intermediate vertex for every transition that is not alternating.

6.2.3 Transformation Version 3: Synchronization

Last, we will look at the third version that is specifically different from the first two versions.
The transformation of the vertices and actions of the SSG is identical to the first two versions. We translate a vertex into a tuple that consists of two variables. For each vertex, one of these variables is zero, indicating to which player the respective state belongs.
The difference to the other versions is that each has two variables in their module instead of one. Both players track at which state the play currently is in their own module.
At every point where from the current state and the action it is not clear what the state will be, the players have to synchronize. Therefore, we add a third variable for each of the players that keeps track of the synchronization.
In this transformation we do not modify the input SSG and therefore directly begin with the transformation to an SMG.
Since the transformation of the vertices and the actions is identical, we refer to Algorithms 6.2.3 and 6.2.4 and assume that we already executed them. We make the same assumptions as in Section 6.2.1 and extend them:

- $probacts_{\exists}(G)$ is a function that returns true if and only if the player Eve has control over a vertex that has an outgoing probabilistic action. The analogous holds for $probacts_{\forall}(G)$ and Adam.

Algorithm 6.2.8 Version 2, Phase 1: Modifying SSG to be alternating

Input: Stochastic Arena $G = (V, V_\exists, V_\forall, v_I, A, Av, \delta)$
Output: Stochastic Arena $G' = (V', V'_\exists, V'_\forall, v'_I, A', Av', \delta')$ which is alternating

1: $G' := G$
2: $A' := A' \uplus \{\text{extra\_eve\_action}\}$
3: $A' := A' \uplus \{\text{extra\_adam\_action}\}$
4: **for all** $t = (v, a, \mu) \in \mathcal{T}(G')$ **do**
5: **if** t is deterministic **then**
6: **if** t is a self-loop **then**
7: **continue**
8: **else if** $v \in V'_\exists$ and $v_{t,1} \in V'_\exists$ **then**
9: $V'_\forall := V'_\forall \cup \{v_{new}\}$ with new vertex v_{new}
10: $\delta'(v, a) := \mu_{new}$ where $\mu_{new}(v_{new}) = 1$
11: $\delta'(v_{new}, \text{extra\_adam\_action}) := \mu_{new}$ where $\mu_{new}(v_{new}) = 1$
12: **else if** $v \in V'_\forall$ and $v_{t,1} \in V'_\forall$ **then**
13: $V'_\exists := V'_\exists \cup \{v_{new}\}$ with new vertex v_{new}
14: $\delta'(v, a) := \mu_{new}$ where $\mu_{new}(v_{new}) = 1$
15: $\delta'(v_{new}, \text{extra\_eve\_action}) := \mu_{new}$ where $\mu_{new}(v_{new}) = 1$
16: **end if**
17: **else**
18: **if** $v \in V'_\exists$ **then**
19: Initialize $\mu_{end} : e(t) \to \mathbb{Q}_{\geq 0}$
20: **for all** $v_e \in e(t)$ **do**
21: **if** $v_e \in V'_\exists$ **then**
22: $V'_\forall := V'_\forall \cup \{v_{new}\}$ with new vertex v_{new}
23: $\delta'(v_{new}, \text{extra\_adam\_action}) := \mu_{new}$ where $\mu_{new}(v_e) = 1$
24: $\mu_{end}(v_{new}) := \mu(v_e)$
25: **else**
26: $\mu_{end}(v_e) := \mu(v_e)$
27: **end if**
28: **end for**
29: $\delta'(v, a) := \mu_{end}$
30: **else**
31: Initialize $\mu_{end} : e(t) \to \mathbb{Q}_{\geq 0}$
32: **for all** $v_e \in e(t)$ **do**
33: **if** $v_e \in V'_\forall$ **then**
34: $V'_\exists := V'_\exists \cup \{v_{new}\}$ with new vertex v_{new}
35: $\delta'(v_{new}, \text{extra\_eve\_action}) := \mu_{new}$ where $\mu_{new}(v_e) = 1$
36: $\mu_{end}(v_{new}) := \mu(v_e)$
37: **else**
38: $\mu_{end}(v_e) := \mu(v_e)$
39: **end if**
40: **end for**
41: $\delta'(v, a) := \mu_{end}$
42: **end if**
43: **end if**
44: **end for**
45: **return** G'

- $probacts_\exists(G)$ and $probacts_\forall(G)$ are computed once and then cached since this result is constant. Therefore, for every appearance after we computed this once, we assume that we use the cached value.
- $probvertex(G, v)$ is a function with $v \in V$ and $G = (V, \ldots)$ that returns true if and only if v has outgoing probabilistic transitions in G.

Now we have a look at the algorithm for Version 3. We split it into several parts again: The differences from Version 1 and Version 2 are highlighted. This also holds for algorithms that we discuss later in this section. In Algorithm 6.2.9, we perform the following steps:

1. In the beginning (Line 1 and 2), we perform Algorithm 6.2.3 and 6.2.4.

Algorithm 6.2.9 Version 3: Create SMG specification (1)

Input: SSG object $(G = (V, V_\exists, V_\forall, v_I, A, Av, \delta), RE(T))$
Output: SMG specification

```
1: perform Algorithm 6.2.3
2: perform Algorithm 6.2.4
3: spec := "smg\n"
4: spec += "player eve\n"
5: spec += "evemod, [e1], ...,[e⟨actcount∃ − 1⟩]"
6: if probacts∃(G) then
7:     spec += ", [ep]"
8: end if
9: spec += "\nendplayer\n"
10: spec += "player adam\n"
11: spec += "adammod, [a1], ..., [a⟨actcount∀ − 1⟩]"
12: if probacts∀(G) then
13:     spec += ", [ap]"
14: end if
15: spec += "\nendplayer\n"
```

Algorithm 6.2.10 Version 3: Create SMG specification (2)

```
1: re = ""
2: ra = ""
3: if probacts∃(G) then
4:     re += " & re=0"
5: end if
6: if probacts∀(G) then
7:     ra += " & ra=0"
8: end if
9: emod = "module evemod\n"
10: emod += "e1 : [0..⟨max(1, statecount∃ − 1)⟩] init ⟨LV(vI)⟩;\n"
11: emod += "e2 : [0..⟨max(1, statecount∀ − 1)⟩] init ⟨RV(vI)⟩;\n"
12: if probacts∃(G) then
13:     emod += "re : [0..1] init 0;\n"
14: end if
15: amod = "module adammod\n"
16: amod += "a1 : [0..⟨max(1, statecount∃ − 1)⟩] init ⟨LV(vI)⟩;\n"
17: amod += "a2 : [0..⟨max(1, statecount∀ − 1)⟩] init ⟨RV(vI)⟩;\n"
18: if probacts∀(G) then
19:     amod += "ra : [0..1] init 0;\n"
20: end if
```

2. We create the player Eve in Line 4-9 as in Algorithm 6.2.5. Additionally, we add an extra action that is used for synchronization in Algorithm 6.2.11. We only do this if Eve has control over a vertex which has an outgoing transition that is probabilistic.

3. We create the player Adam in Line 10-15 and add an extra action for Adam if necessary, analogously to Eve.

Since we enforce that the input SPG is stochastic and not deterministic, in a full reduction process in Stargate, we always add at least one of the extra actions. In Algorithm 6.2.10 we create the modules, which comprises the following actions:

1. In Lines 1-8, we create empty strings "re" and "ra" for each player, respectively. They are updated with a string containing an extension of a guard when $probacts_\exists(G)$ or $probacts_\forall(G)$ is true, respectively. We use these strings to account for whether the transitions of the players need to be further synchronized. If not, then we simplify the case for deterministic transitions where there is no need for synchronization.

2. We do the actual creation of the modules for both players in Lines 9-20. We have two variables `e1`, `e2` and `a1`, `a2` for Eve and Adam, respectively, that represent the state of the SMG. The next part of Algorithm 6.2.11 ensures the synchronization between `e1` and `a1`, as well as between `e2` and `a2`.

Algorithm 6.2.11 Version 3: Create SMG specification (3)

1: **for all** $t = (v, a, \mu) \in \mathcal{T}(G)$ **do**
2: **if** $v \in V_\exists$ **then**
3: **if** not $probvertex(G, v)$ **then**
4: emod += "`[`$\langle act_\exists(a)\rangle$`] (e1=`$\langle LV(v)\rangle$ `& e2=`$\langle RV(v)\rangle\langle \text{re}\rangle$`) -> ` " +
5: "`(e1'=`$\langle LV(v_{t,1})\rangle$`) & (e2'=`$\langle RV(v_{t,1})\rangle$`);`\n"
6: amod += "`[`$\langle act_\exists(a)\rangle$`] (a1=`$\langle LV(v)\rangle$ `& a2=`$\langle RV(v)\rangle\langle \text{re}\rangle$`) -> ` " +
7: "`(a1'=`$\langle LV(v_{t,1})\rangle$`) & (a2'=`$\langle RV(v_{t,1})\rangle$`);`\n"
8: **else**
9: **if** $|e(t)| = 1$ **then**
10: emod += "`[`$\langle act_\exists(a)\rangle$`] (e1=`$\langle LV(v)\rangle$ `& e2=`$\langle RV(v)\rangle$ `& re=0) -> ` " +
11: "`(e1'=`$\langle LV(v_{t,1})\rangle$`) & (e2'=`$\langle RV(v_{t,1})\rangle$`) & (re'=1);`\n"
12: **else**
13: emod += "`[`$\langle act_\exists(a)\rangle$`] (e1=`$\langle LV(v)\rangle$ `& e2=`$\langle RV(v)\rangle$ `& re=0) -> ` "
14: **for all** $v_e \in e(t)$ **do**
15: emod += "`(`$\langle \mu(v_e)\rangle$`) : (e1'=`$\langle LV(v_e)\rangle$`) & (e2'=`$\langle RV(v_e)\rangle$`) & (re'=1) + ` "
16: **end for**
17: emod = (emod - " `+` ") + "\n"
18: **end if**
19: amod += "`[`$\langle act_\exists(a)\rangle$`] (a1=`$\langle LV(v)\rangle$ `& a2=`$\langle RV(v)\rangle$ `& re=0) -> true;`\n"
20: **end if**
21: **else**
22: **if** not $probvertex(G, v)$ **then**
23: amod += "`[`$\langle act_\forall(a)\rangle$`] (a1=`$\langle LV(v)\rangle$ `& a2=`$\langle RV(v)\rangle\{\text{ra}\}$`) -> ` " +
24: "`(a1'=`$\langle LV(v_{t,1})\rangle$`) & (a2'=`$\langle RV(v_{t,1})\rangle$`);`\n"
25: emod += "`[`$\langle act_\forall(a)\rangle$`] (e1=`$\langle LV(v)\rangle$ `& e2=`$\langle RV(v)\rangle\{\text{ra}\}$`) -> ` " +
26: "`(e1'=`$\langle LV(v_{t,1})\rangle$`) & (e2'=`$\langle RV(v_{t,1})\rangle$`);`\n"
27: **else**
28: **if** $|e(t)| = 1$ **then**
29: amod += "`[`$\langle act_\forall(a)\rangle$`] (a1=`$\langle LV(v)\rangle$ `& a2=`$\langle RV(v)\rangle$ `& ra=0) -> ` " +
30: "`(a1'=`$\langle LV(v_{t,1})\rangle$`) & (a2'=`$\langle RV(v_{t,1})\rangle$`) & (ra'=1);`\n"
31: **else**
32: amod += "`[`$\langle act_\forall(a)\rangle$`] (a1=`$\langle LV(v)\rangle$ `& a2=`$\langle RV(v)\rangle$ `& ra=0) -> ` "
33: **for all** $v_e \in e(t)$ **do**
34: amod += "`(`$\langle \mu(v_e)\rangle$`) : (a1'=`$\langle LV(v_e)\rangle$`) & (a2'=`$\langle RV(v_e)\rangle$`) & (ra'=1) + ` "
35: **end for**
36: amod = (amod - " `+` ") + "\n"
37: **end if**
38: emod += "`[`$\langle act_\forall(a)\rangle$`] (e1=`$\langle LV(v)\rangle$ `& e2=`$\langle RV(v)\rangle$ `& ra=0) -> true;`\n"
39: **end if**
40: **end if**
41: **end for**
42: **if** $probacts_\exists(G)$ **then**
43: emod += "`[ep] (re=1) -> (re'=0);`\n"
44: amod += "`[ep] (re=1) -> (a1'=e1) & (a2'=e2);`\n"
45: **end if**
46: **if** $probacts_\forall(G)$ **then**
47: amod += "`[ap] (ra=1) -> (ra'=0);`\n"
48: emod += "`[ap] (ra=1) -> (e1'=a1) & (e2'=a2);`\n"
49: **end if**
50: spec += emod + "`endmodule`\n" + amod + "`endmodule`\n"

We require the variable `re` if at least one vertex controlled by Eve has an outgoing probabilistic transition, and `ra` if this is the case for a vertex controlled by Adam. We use these variables as flags in the way that they have value 1 if the other variables `e1`, `e2`, `a1`, and `a2` need to be synchronized.

In Algorithm 6.2.11, we execute the following actions:

1. In the for-loop starting in Line 1, we iterate over all transitions $\mathcal{T}(G)$ in G.
   Without loss of generality we assume that the transition we examine is controlled by Eve. The case for Adam is analogous.

Algorithm 6.2.12 Version 3: Create SMG specification (4)

```
1: spec += "label "target" = ("
2: for all v ∈ T do
3:     spec += "(e1=⟨LV(v)⟩) & (e2=⟨RV(v)⟩) | "
4: end for
5: if exists string s' with s' + "(" = spec then
6:     spec = (spec - "(") + "false;"
7: else
8:     spec = (spec - " | ") + ");"
9: end if
10: return spec
```

1.1. If the start vertex v has no outgoing transitions that are probabilistic, we add a guarded command to each module (Lines 4-7). If this case occurs, then the transition we are currently transforming is deterministic as well.
The guards of these commands ensure that the current values of the state variables match the state variables of the starting vertex of the transition. If "re" is not an empty string, we need to consider synchronization. Then the string "re" states that we require the variable `re` to be 0 which means that we are not in a synchronization process.

1.2. If the start vertex v has at least one outgoing transition that is probabilistic, we distinguish between deterministic and probabilistic transitions.

If the transition is deterministic, we add a guarded command to Eve's module (Line 10). To perform the updates in this guarded command, we require to be in the starting state. The updates ensure that afterwards we are in the end state of the transition. Additionally, we set a flag for synchronization.

If the transition is probabilistic, we add a guarded command that represents the probability distribution of the transition's end vertices in Lines 13-17. As in the other case, we set the synchronization flag `re`.

Regardless of whether the transition is deterministic or probabilistic, we add a guarded command to Adam's module that means that he waits for the synchronization since we are in a state where synchronization needs to be performed (Line 19).

2. In Lines 42-49 we add the guarded commands that synchronize the variables of the modules. This is done for Eve and Adam if $probacts_{\exists}(G)$ and $probacts_{\forall}(G)$ are true, respectively. We examine the commands for Eve.
After Eve performs an action that needs to be synchronized, she changes the flag's value (`re`) back to 0. Simultaneously, Adam copies the value of Eve's variable.
Assume that the probabilistic guarded command that is added for Eve in Lines 13-17 is also added for Adam and this command also contains a probabilistic decision. Then there is the chance that Adam reaches a different end vertex's state than Eve. With this, the SSG construction becomes inconsistent. This is why we need to copy the value.

3. Finally, in Line 50, the modules are attached to the specification string.

The labeling in the last part of Version 3 (Algorithm 6.2.12) is almost identical to the labeling of the first two versions (Algorithm 6.2.7). The parts that differ are highlighted.
The only difference is the name of the state variables. For this we decided for Eve's variables but Adam's would semantically be equal since they are synchronized.

6.3 Use of PRISM-games extension

Next, we present how STARGATE uses the PRISM-games extension presented in [18] to solve the SMGs resulting from the transformation. In the following section, when we use the term PRISM or PRISM-games we refer to the extension proposed by Křetínský et al. in [18].

Overview: We use an SMG specification resulting from the SSG transformation (see Section 6.2) as input. We save this specification to a file since PRISM-games only operates on models that are input via a file.

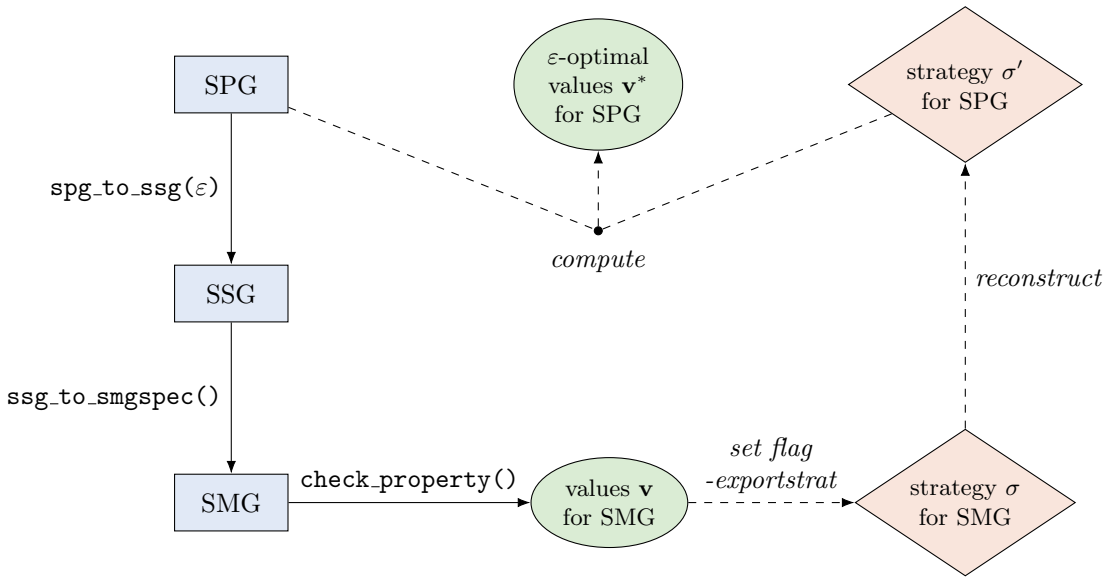


Figure 5: Illustration of SPG solving process with STARGATE.
Solid lines represent actions in STARGATE. Dashed lines are features dependent on -exportstrat flag that is currently not implemented in the extension of PRISM-games.

After saving, STARGATE starts a PRISM-games subprocess that solves the SSG. We illustrate the idea behind this solving process in Figure 5.
With PRISM-games we solve the game quantitatively to obtain values $\mathbf{v}$ with respect to the target reachability in the SMG. With the flag *-exportstrat*, we would receive a strategy σ for the SMG. Unfortunately, in the extension we use, this option is not available. We assume that this option existed:
With the resulting strategy σ we can trace back and reconstruct a strategy σ' for the SPG.
With σ' and the SPG from the beginning of the reduction we are able to calculate ε-optimal values $\mathbf{v}^*$ for the SPG. When we use the ε given in [3] for the reduction from SPG to SSG, then σ' is the optimal strategy, thus the resulting values $\mathbf{v}$ can be computed exactly.
All these possible future features are represented with dashed lines in Figure 5.

PRISM-games in STARGATE: STARGATE implements four methods that use PRISM-games:

- **check_property():** This method uses the PRISM-games command line interface to solve the input SMG for a property that is based on rPATL [8]. The user can set several flags for this process, e.g., solving algorithm, maximum iterations and ε indicating how precise the result from PRISM-games should be.
  In STARGATE, we mainly use this method as a submethod of check_target_reachability(). This method returns values for the property. With these values we could compute optimal strategies following Figure 5.

- **check_target_reachability():** This method is the most significant for solving SPGs in STARGATE. It uses the previously discussed method check_property() twice.

  1. The method lets PRISM-games compute the value and the respective strategy for the property specified as"`<<eve>> Pmin=? [F "target"]`". This asks for the probability that Adam manages to reach a target state in the SMG when Eve tries to prevent this. The value is obtained by computing the respective strategy. With respect to the input SPG, this computes a strategy for Adam to win with an even parity winning objective which we rarely consider for our reduction.
  2. The method solves for the property "`<<eve>> Pmax=? [F "target"]`". This property specifies the probability of Eve winning with an even parity objective. The strategy we obtain translates to a strategy in the SPG to maximize Eve's winning probability with even parity.

  We emphasize that the probabilities that result from these computations are typically close to the values of the SPG.

- **check_smg_stats():** This method is mainly used for validation and runtime analysis which we discuss in Section 7. It starts a PRISM-games subprocess that builds the respective SMG and returns the number of states, number of transitions, and time to build for the SMG.
- **create_dot_file():** This method uses the DOT file export flag from PRISM-games to enable creating graphic formats, such as PNG or SVG, of the SMG.

7 Empirical Evaluation

In this section we examine the tool in three different aspects:

1. Comparison of different SSG to SMG transformation versions from Section 6.2. Here we compare the performance of the three different versions.
2. Correctness checks of Stargate. Here we test on custom examples and use cases if Stargate computes the right values.
3. Scalability of Stargate. Here we examine the performance and scalability of Stargate in terms of both size and time.

All validation checks ran on a Microsoft Surface with an Intel Core i7-1185G7 @ 3.00GHz CPU with four cores, and 32GB RAM.
We created our own benchmarks for all evaluations.[1] We provide Tables containing detailed results of the experiments in Appendix A.1, A.2, and A.3.

7.1 Comparison of SSG to SMG Transformation Versions

We start with the comparison of the SSG to SMG transformation versions.
Our approach of comparing Versions 1, 2, and 3 of the transformation is that we take Version 1 and compare Versions 2 and 3 to it, respectively.
We compare both versions on number of states and transitions of the resulting SMG, and on transformation and property checking time with PRISM-games.

7.1.1 Benchmarks

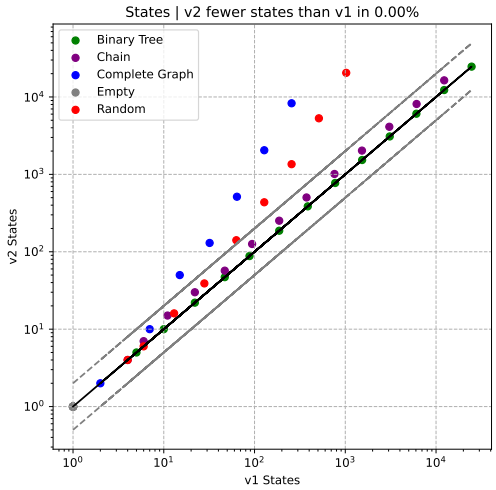
We create different types of SSGs. For each type, we start with a small SSG and double the number of vertices in each iteration. In each iteration, after the creation, we transform the created SSG into an SMG using the versions we compare. Finally, we solve these SMGs using PRISM-games. For all these steps, we track transformation and solving times as well as the size of the SMG. If one iteration exceeds the timeout limit set to 20 minutes, we continue with the next type of SSG. In total, we check five different types of SSGs:

1. **Random SSGs:** Each vertex is randomly controlled by Eve or Adam. We randomly choose about a tenth of the vertices to be target vertices. Each vertex has a transition to a tenth of the other vertices.
2. **Binary Tree SSGs:** In each iteration we double the number of layers of the binary tree which is analogous to doubling the number of vertices each iteration. Each vertex is randomly controlled by Eve or Adam. The binary tree structure is achieved by probabilistic transitions from parent vertices to two child vertices. 30% of the leaf vertices are target vertices.
3. **Chain SSGs:** The vertices, randomly controlled by Eve or Adam, form a line connected by probabilistic transitions where for a probability of $\frac{1}{2}$ the successor vertex in the chain is reached and with $\frac{1}{2}$ the first vertex of the chain is reached. The vertex at the end of the chain is a target vertex and a sink.
4. **Complete Graph SSGs:** Each vertex is randomly controlled by Eve or Adam and has deterministic transitions to every other vertex. One tenth of the vertices are target vertices.
5. **Empty SSGs:** Each vertex is randomly controlled by Eve or Adam. We only have self-loops at every vertex which we add because PRISM-games requires vertices to have at least one outgoing transition.

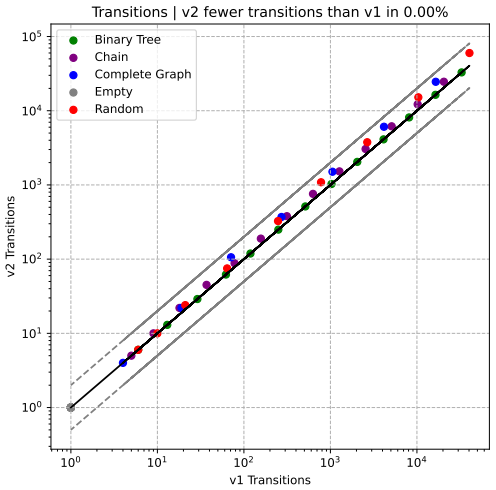
7.1.2 Results

Version 1 vs. Version 2: Figure 6 depicts the comparison of Version 1 and Version 2 concerning the size of the resulting SMG.
The x-coordinate describes the performance of Version 1 and the y-coordinate the performance of Version 2. The solid black line in the middle of each plot represents the two versions performing equally. If a point is above this line, then Version 1 is better for this SSG than Version 2. If a point

[1] To the best of our knowledge, nobody published a benchmark set.

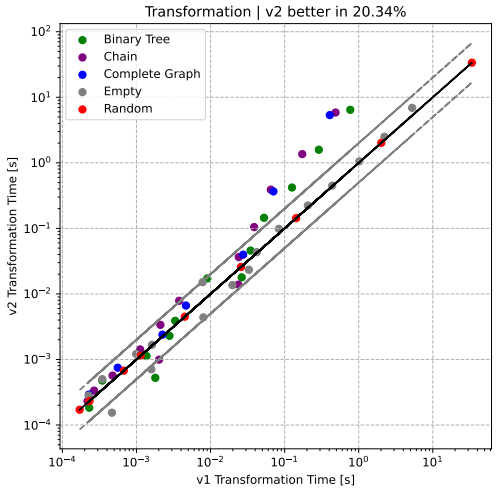


(a) Number of SMG states comparison.

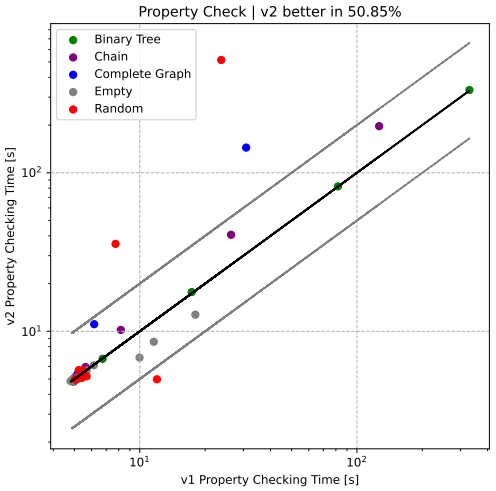


(b) Number of SMG transitions comparison.

Figure 6: Comparison of SMG stats for Version 1 and Version 2 (for data table see Appendix A.1.1).



(a) Transformation time comparison.

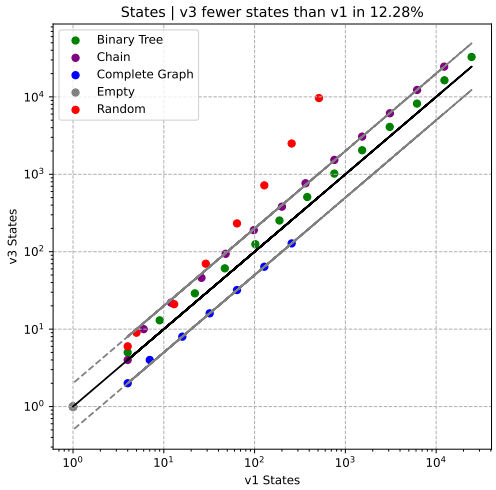


(b) Property checking time comparison.

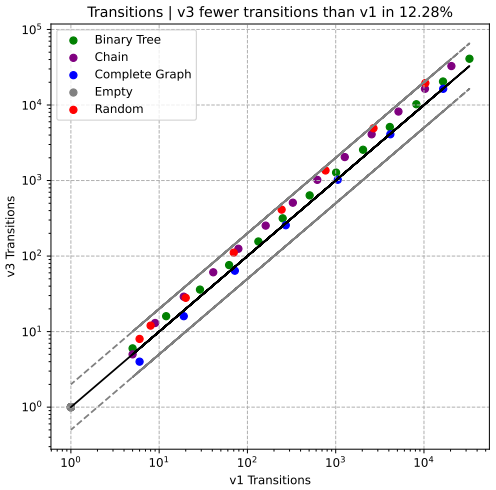
Figure 7: Comparison of runtimes for Version 1 and 2 (for data table see Appendix A.1.2).

is below this line, then Version 2 is better than Version 1 for this SSG. The gray lines in parallel to the equal line means that the value of one version for the examined metric is double the value of the other version.
The different colors of the data points represent the different SSG types as specified in the legend. All axes are logarithmic which holds for all plots in this section.
Figure 6a shows the difference in number of states in the SMG that results after transforming SSGs with Version 1 and Version 2, respectively. We observe that PRISM-games condenses the empty SSGs into SSGs with one state, as every state except the initial one is not reachable in the SMG. The distribution of data points indicates that Version 1 in total creates fewer additional states which we hypothesized in Section 6.2. In particular, Version 1 is better for SSGs where we have many vertices with many incoming transitions. This is where we generate fewer additional states in Version 1 compared to Version 2.
Figure 6b indicates no meaningful difference in resulting number of transitions. Still, there is a tendency of Version 1 performing better as we generate fewer transitions. This leads to a faster analysis of the resulting SMG.

In Figure 7, we depict the comparison of Version 1 and Version 2 concerning transformation and property checking times.
In the plot in Figure 7a, we see that both versions perform about equally when the transformation time is less than 0.1 seconds. When the transformation requires more time, then Version 1 is faster than Version 2. Version 1 was for some instances about 8 times faster than Version 2.

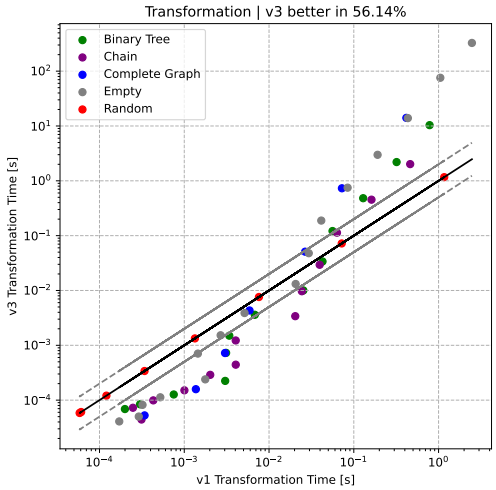


(a) Number of SMG states comparison.

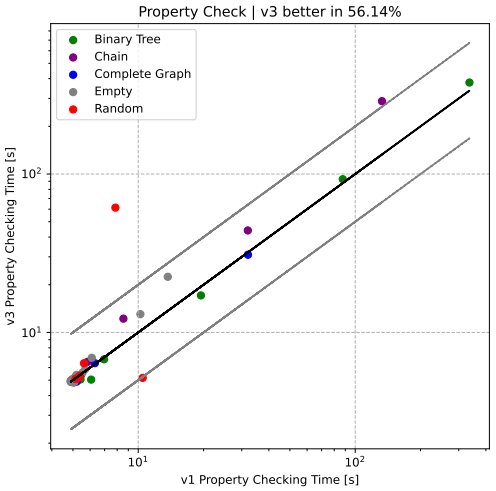


(b) Number of SMG transitions comparison.

Figure 8: Comparison of SMG stats for Version 1 and Version 3 (for data table see Appendix A.2.1).



(a) Transformation time comparison.



(b) Property checking time comparison.

Figure 9: Comparison of runtimes for Version 1 and Version 3 (for data table see Appendix A.2.2).

Regarding the checking time in Figure 7b, no version clearly outperforms the other. We can see that a lot of SMGs took an equally small amount of time to check. There are some points above the line, which indicates that Version 1 might perform better on larger SSGs.

In total, we observe a better performance of Version 1 in some aspects while we see an equal performance in the rest. This dominance is more prominent for SSGs that have a lot of transitions, e.g., complete graph SSGs. In this case, Version 1 only adds a new vertex for each original vertex while Version 2 adds one for each transition that is not alternating.
We conclude that Version 1 performs better than Version 2 in total.

Version 1 vs. Version 3: The plots in Figure 8 are structured as the ones in Figure 6.
The plot in Figure 8a indicates close numbers of states for smaller SSGs between Versions 1 and 3. For bigger SSGs, we see that Version 1 creates fewer states than Version 3. This is not the case for complete SSGs. We suspect that this stems from the fact that complete SSGs only have deterministic transitions. In non-synthetic use cases for STARGATE this should never occur, since we require the SPG to be stochastic.
Figure 8b shows that the resulting number of transitions in the SMG is about equal, same as in the comparison between Version 1 and 2. There is a slight tendency towards fewer transitions with Version 1.

In Figure 9 we see two plots showing the comparison of Version 1 and Version 3 regarding trans-

formation time and property checking time.
Figure 9a reveals a notable pattern. The transformation time is better for small SSGs with Version 3. However, Version 1 becomes better with increasing size where Version 1 is up to about 300 times better for our SSGs.
The property checking time in Figure 9b indicates no clear domination of one of the versions. Most data points are accumulated with short times on the equal line. For bigger SSGs, there is a tendency towards Version 1 performing better.

Similar to the comparison of Version 1 and Version 2, Version 1 performs better in certain aspects. In the other aspects, Version 1 and 3 were about equal. Therefore, we conclude that overall Version 1 tends to perform better than Version 3.

7.2 Validating Stargate

In this section, we validate Stargate by checking correctness and performance with respect to output size and computation time.
We decide not to compare Stargate to other tools as none are currently available.[2]
We first examine the correctness of the tool.

7.2.1 Correctness

In order to check correctness, we run Stargate on the following custom benchmarks:

1. Small
2. Chain
3. ModifiedMutex
4. FrozenLake

Benchmarks: We first discuss the benchmark SPGs.

Small SPGs: The SPGs we check first are very similar.
We specify the first SPG in SPLANG and show its a graph in Figure 10.
We represent two SPGs at once. SPG1 is defined by the whole specification in SPLANG on the left. SPG2 is defined by the specification in SPLANG on the left where the transition that is highlighted in blue is omitted.
We choose these SPGs as they have properties that test Stargate's capabilities such as multiple outgoing transitions, multiple priorities for both players, and deterministic and probabilistic transitions.

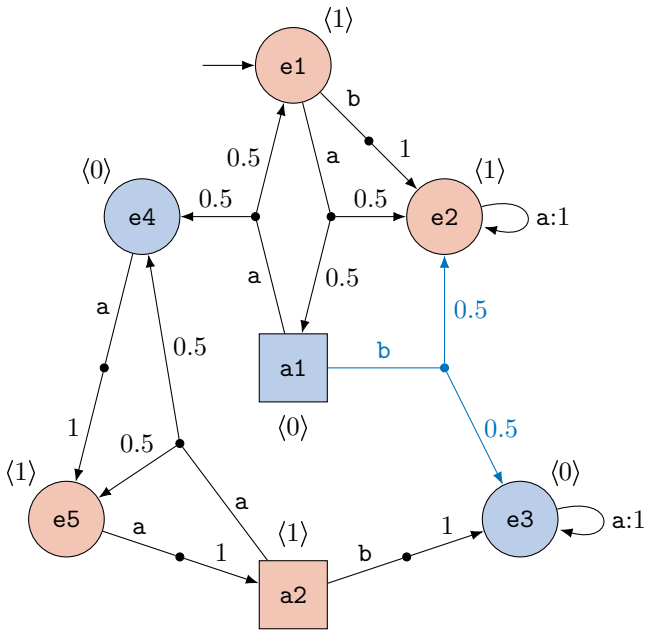
Chain SPGs: We check the correctness of Stargate on chain SPGs. The structure of a chain SPG is outlined in Figure 11. Such an SPG is constructed such that for length l, we create $l+1$ Eve vertices $v_0, \ldots, v_l$. Each vertex $v_i, i \in \mathbb{N}_{\leq l-1}$ has priority 0 and an outgoing a transition with probability $\frac{1}{2}$ to reach v_{i+1} and with probability $\frac{1}{2}$ to reach v_0. Only v_l has a self-loop and priority 0. Eve winning with even parity is equivalent to a play reaching v_l.

ModifiedMutex SPG: Another SPG we test Stargate on is a modified mutex scenario.
In the SPG, we model the following behavior:
Both players Eve and Adam control a process. Each process can be in state neutral (N) or critical (C). Eve tries to achieve that infinitely often exactly one process is in C and not infinitely often both are in C. When a process is in N and it is its turn to act then the process can decide whether to go into the critical section, i.e., set his status to C, or stay in N. When in C, the process leaves C and is in N after performing the next action. Which process's turn it is, is chosen at random with 0.5 for Eve and 0.5 for Adam to act.
In the SPG this is realized by a state space of $\{N, C\}^2 \times \{0, 1\}$. The first field represents the state of Eve's process while the second represents the state of Adam's process. The last field is 0 when the next turn Eve takes action and analogously for 1 with Adam.
The states $(N, N, 0)$ and $(N, N, 1)$ have priority 3, $(N, C, 0)$, $(N, C, 1)$, $(C, N, 0)$, and $(C, N, 1)$ have priority 2. $(C, C, 0)$ and $(C, C, 1)$ have priority 1. Eve achieving her goal translates to her winning the SPG with even parity.

[2] While the tool GIST [7] is a possible candidate, we were not able to obtain it, even after contacting its developers.

```
spg

evevertices
e1 : 1
e2 : 1
e3 : 0
e4 : 0
e5 : 1
endevevertices

adamvertices
a1 : 0
a2 : 1
endadamvertices

initialvertex : e1

transitions
e1 a : 0.5 | a1 + 0.5 | e2
e1 b : e2
e2 a : e2
a1 a : 0.5 | e1 + 0.5 | e4
a1 b : 0.5 | e3 + 0.5 | e2
e4 a : e5
e5 a : a2
a2 a : e3
e3 a : e3
a2 b : 0.5 | e5 + 0.5 | e4
endtransitions
```

(a) SPG specification in SPLANG.



(b) SPG graph.

Figure 10: SMALL SPG specification (left) and its corresponding graph (right). Vertices with priority 0 are blue, vertices with priority 1 are red.

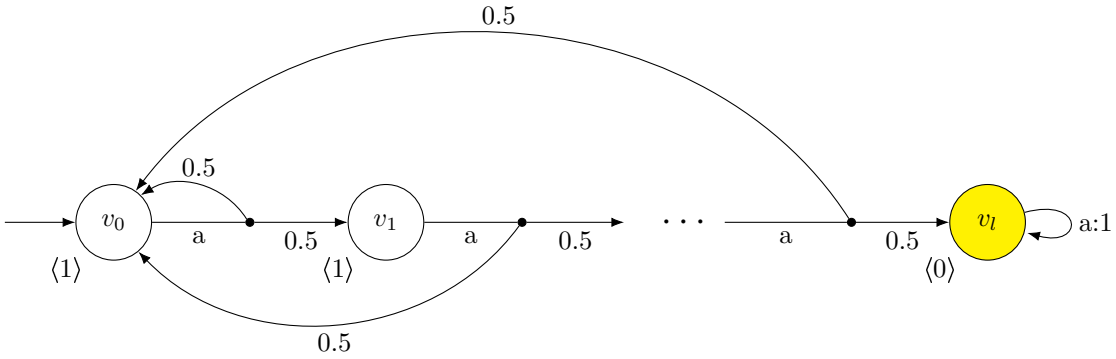


Figure 11: Chain SPG of length l. Vertices are yellow with priority 0 and white with priority 1.

FROZENLAKE SPGs: The last benchmark for correctness we perform is FROZENLAKE which is based on the Frozen Lake presented by Towers et al. [26].
FROZENLAKE describes the scenario where a player (in our case Eve) controls an agent we call Jim that is walking on a rectangular grid on a frozen lake with several holes in the ice. Since the lake is frozen, it is slippery and it can happen with a certain probability that Jim slides one cell further than he intended to. Jim cannot slip or get blown off the lake.
Additionally, it is windy and with a certain probability after a move of Jim, the wind (controlled by Adam) actively tries to blow Jim into one of the holes in the lake. Jim tries to alternatingly reach two different target cells on the ice. Eve wins if Jim manages to achieve his goal of reaching the target cells infinitely often while only finitely often falling into a hole.
In the implementation, we modeled the frozen lake by a state space that is given by the cell Jim is currently on, which target Jim has to go next, and if the next turn is Jim's (Eve's) or the wind's (Adam's). The target that Eve tries to reach next has priority 2 and switches to 3 if reached and the other target changes to priority 2. All normal cells have priority 3 and the holes have priority 1.

SPG Benchmark	Expected Value	Computed Value by STARGATE
SMALL		
SPG1	0.25	0.2500000002441407
SPG2	$0.\overline{33}$	0.33333301565891216
CHAIN		
Chain (Length 2)	1.0	0.9999964848087903
Chain (Length 4)	1.0	0.9999734628674462
Chain (Length 8)	1.0	0.9994974341306767
Chain (Length 16)	1.0	0.884130763557911
Chain (Length 32)	1.0	0.00011640179496398849
Chain (Length 64)	1.0	0.0
MODIFIEDMUTEX		
Mutex	0.0	$1.1920962035649514 \cdot 10^{-14}$
FROZENLAKE		
Lake (1 row, 3 columns)	1.0	$3.919220332322857 \cdot 10^{-8}$

Table 1: Comparison of expected and computed values on SPG benchmarks for correctness evaluation.

Results: We discuss the results for every correctness SPG benchmark. For every SPG to SSG transformation, we applied the reduction with ε-optimality where $\varepsilon = 10^{-6}$. The resulting values, as well as the expected values for each benchmark, are given in Table 1.

SMALL SPGs: For SPG1, we observe that Eve has a winning probability of 0.25. STARGATE computes a winning probability of 0.2500000002441407, which is a good result considering $\varepsilon = 10^{-6}$.
For SPG2 that we obtain by omitting the highlighted transition in SPG1, we observe that Eve has a winning probability of $0.\overline{33}$. With STARGATE, we obtain a probability of 0.33333301565891216 which is equally good as for SPG1.

Therefore, STARGATE computed the correct probabilities on these two examples.

CHAIN SPGs: Winning for Eve with even parity is equivalent to a play reaching the last vertex of the chain which is 1 independent of the chain's length.
For length $l = 2$, STARGATE computes a value of approximately 1 within acceptable precision. For $l = 4$, and 8, our tool computes a winning probability of approximately 1 but with decreasing precision. For $l = 16$, the value is close to 0.88. For all higher lengths, the value is approximately 0.
This shows that the winning probabilities quickly assume incorrect values. We observed in the graphs of the resulting SMGs that the SMG is correctly generated. Therefore, the imprecision lies in the solving process by PRISM-games caused by small probabilities in the SMGs. The alpha values for $l = 32$ are about 10^{-17} for $\alpha(0)$ and about 10^{-34} for $\alpha(1)$. This imprecision for small numbers by probabilistic model checkers is a known limitation [27].

MODIFIEDMUTEX SPG: We obtain a probability for Eve to win this modified mutex scenario of 0. This is because in the limit Adam will with probability 1 infinitely often violate the mutex constraint.
STARGATE computes a winning probability for Eve of $1.1920962035649514 \cdot 10^{-14}$ which is approximately 0. Thus, the modified mutex example works in STARGATE.

FROZENLAKE SPGs: STARGATE never computes the right probability for this scenario. We built a minimal frozen lake with a 1 by 3 grid and both target cells on the outer fields and no holes. Intuitively, the probability of Eve winning this with even parity should be close to 1, but STARGATE returns a value close to zero even for this very small example. From analysis of the graph of the resulting SMG, we learned that with optimal strategies the probability for Eve to win indeed is 1.

Therefore, we suspect the imprecision stems from PRISM-games solving the SMG, similar to the Chain SPGs.

7.2.2 Scalability

As we saw in Section 7.2.1, Stargate struggles with computing the correct values for larger SPGs. We suspect that this imprecision stems from PRISM-games in combination with small transition probabilities resulting from the alpha values. Therefore, it is still reasonable to check the performance of Stargate in terms of output size and transformation time to assess its scalability in case Stargate is adapted for a different solver in the future.
We first present the benchmark set and subsequently discuss the results of the scalability evaluation.

SPG Benchmark set: We first randomly generate a set of SPG benchmarks. For this, we iterate over the following parameters:

- **Number of vertices:** $\{2, 3, 8, 15, 32, 63, 128, 255, 512, 1023\}$
  Whether a vertex is controlled by Eve or Adam is chosen at random.
- **Share of outgoing transitions:** $\{0.05, 0.1, 0.2, 0.5, 1.0\}$
  This represents the number of outgoing transitions for every vertex such that a share of outgoing transitions of x means $\lceil(\text{number of vertices} \cdot x)\rceil$ outgoing transitions per vertex. If a transition is deterministic or probabilistic is randomly chosen. The end vertices of each transition are chosen at random as well.
- **Number of priorities:** $\{2, 4, 8, 32\}$
  The priority is assigned at random for each vertex.

Combinations that are unreasonable such as more priorities than vertices are omitted. Additionally, for combinations that result in the same actual SPG parameters such as share of outgoing transitions of 0.1 and 0.2 for number of vertices = 2, only one SPG is created.
In total, our benchmark set comprises 152 SPGs.
On this set, we check every configuration of transformation for all $\varepsilon \in \{None, 10^{-6}, 10^{-2}, 10^{-1}\}$ (where $None$ means that we perform the SPG to SSG reduction for optimal strategies) and PRISM-games algorithms Value Iteration and Policy Iteration.
We consider the transformation time for the complete transformation from SPG to SMG in seconds, the size of the resulting SMG in bytes, and for the algorithms the time that the property check takes in seconds. For the transformation we do not set a timeout. For the property check, the timeout is 600 seconds.
Note that we do not consider the resulting values of the SPGs.

Results: We evaluate the scalability of Stargate with respect to SMG size in bytes, and transformation time in seconds. Afterwards, we compare Value Iteration and Policy Iteration with respect to property checking time.
All plots have logarithmic axes. For detailed data see Appendix A.3.

SMG Size: Since, the number of vertices for each of the SPG is one of {2, 3, 8, 15, 32, 63, 128, 255, 512, 1023}, the data points in Figure 12 form several columns where each column corresponds to one number of vertices. The data points in each data point column form clusters. One data point cluster corresponds to one combination of number of vertices and share of outgoing transition. This shows that for a fixed number of vertices and transitions, the ε for the transformation barely influences the size of the resulting SMG.
The size appears to scale almost linearly with size, at least for the range of sizes that we checked Stargate on. This is very good and indicates feasible scaling regarding space resources.

Transformation Time: The plot in Figure 13 illustrates the scaling of transformation time. It has a similar structure to Figure 12 with columns of data points. Figure 13 differs in the structure of the columns. For smaller SPGs, these are not clustered. However, as the number of vertices increases, clusters begin to emerge again. This indicates that ε loses influence with a growing number of vertices. There is no domination regarding time of one of the ε values. The distribution of the data points suggests a similar scalability in time compared to size which signifies that with a growing number of vertices, the transformation time scales well.

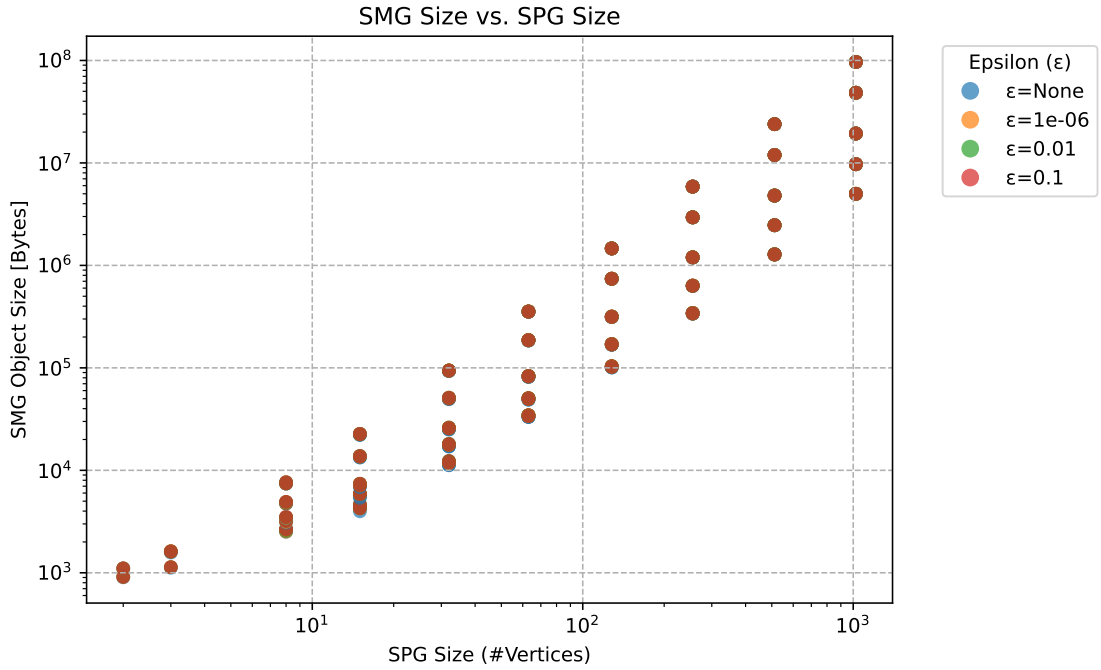


Figure 12: Correlation between number of vertices in the SPG and size of resulting SMG from transformation in STARGATE.

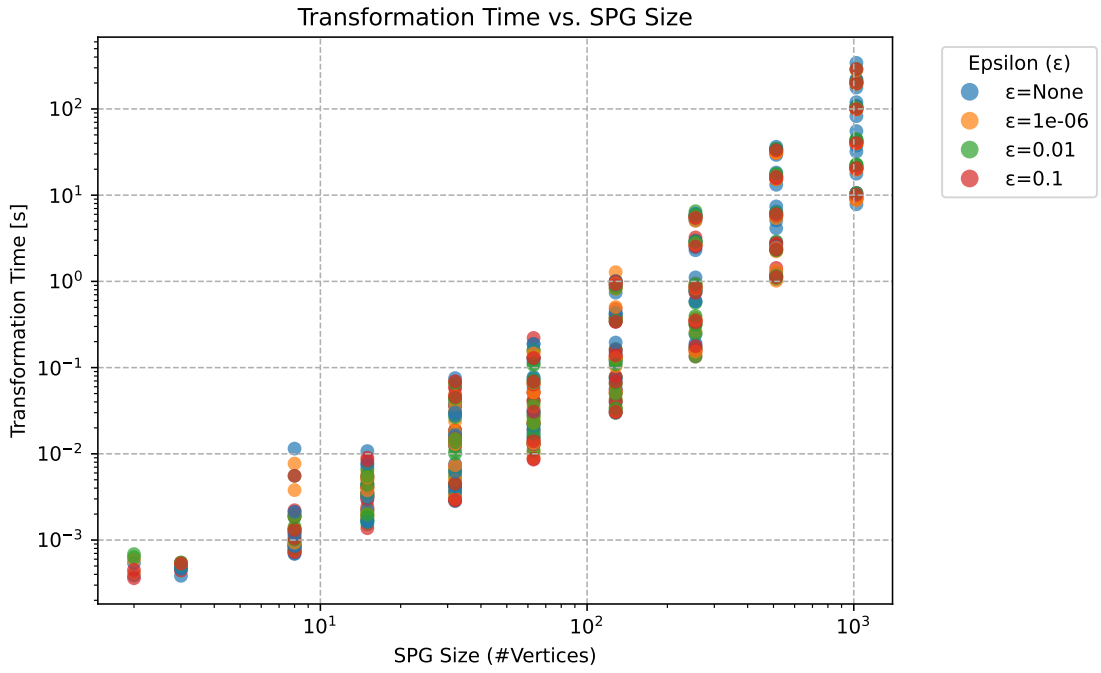


Figure 13: Correlation between number of vertices in the SPG and transformation time from SPG to SMG in STARGATE.

Value Iteration vs. Policy Iteration: The plot in Figure 14 shows the comparison of the property checking time between Value Iteration and Policy Iteration.
All data points beyond the vertical or horizontal red dotted line indicate an error when performing Value Iteration or Policy Iteration, respectively. Regarding timeouts, there are only a few cases where Policy Iteration times out and Value Iteration does not. We observed no pattern in these SMGs of why the timeout is reached.
It appears that for big SPGs with more than 100,000 transitions errors occur in PRISM-games when checking the property. When this error appears, it is for both Value Iteration and Policy Iteration.

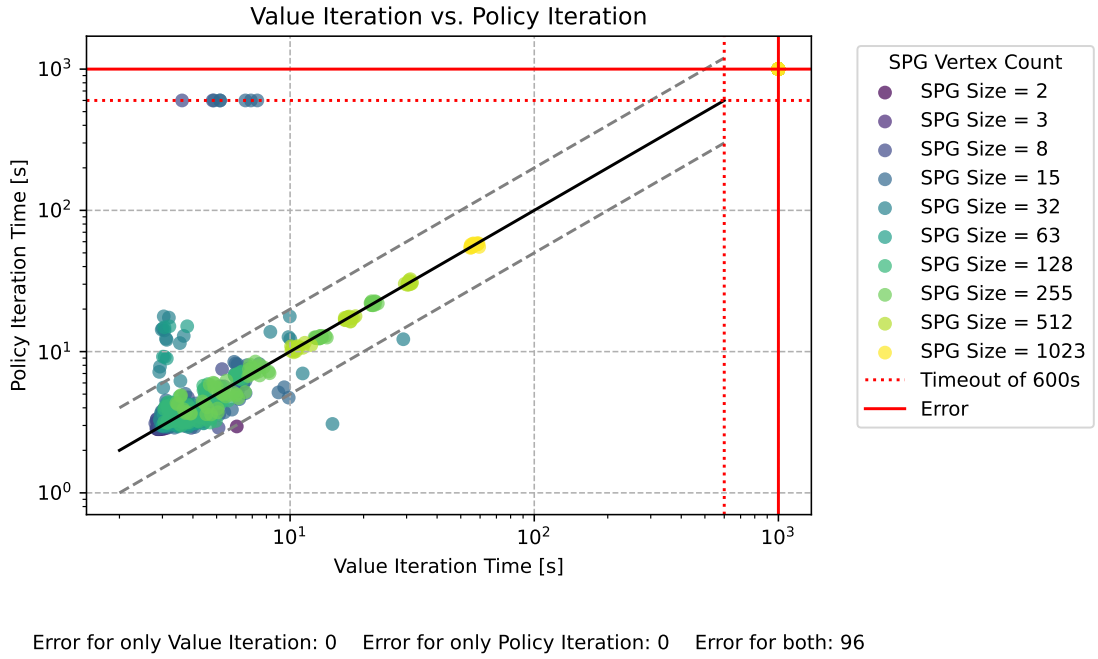


Figure 14: Comparison of property checking time with Value Iteration and Policy Iteration. Points on the red dotted lines signal timeouts, and points on the red solid lines signal errors.

That leads to the assumption that the solving process already fails in the model building phase. Finding a solution to this occurring error in PRISM-games is out of scope for this work.
The plot shows that when the property checking process successfully terminates, the checking time is often under 10 seconds.
Ultimately, the plot indicates that no algorithm clearly outperforms the other. Therefore, we conclude that they are about equally performing.

8 Conclusion

In this paper, we gave an overview of the tool Stargate that solves SPGs by reducing them to SSGs and subsequently solving them using PRISM-games. Afterwards, we introduced the two specification languages SPLANG and SSLANG, which allow the user to load SPGs and SSGs from specification files.

We examined the reduction and gadget construction proposed by Berthon et al. [3], for transforming SPGs into SSGs. We extended this reduction to support ε-optimality by adapting the computation of the alpha values. These alpha values represent the winning probabilities of the input SPG in the resulting SSG. They are probabilities in the SSG to finish the play by reaching one of the game deciding vertices, either v_{win} or v_{lose}. This extension is necessary to address numerical precision issues for very small alpha values, which tend to be rounded to zero when using floating-point representations.

To solve the resulting SSGs, we selected PRISM-games as the solver. Accordingly, we designed three transformation strategies to convert SSGs into SMGs compatible with PRISM-games. We developed two variants based on alternating vertices and one using intermediate states to coordinate player variables.

In order to solve the resulting SMGs, we integrated an extended version of PRISM-games into Stargate. This was crucial as Value Iteration alone often fails to converge. A core limitation of PRISM-games is its inability to perform exact arithmetic operations. This poses a challenge for Stargate, particularly when dealing with very small alpha values that arise from high vertex counts, numerous priorities, or small ε parameters in the reduction. These numerical issues became more apparent during correctness validation where even moderately sized SPGs led to inaccurate results.

Excluding the limitations of PRISM-games itself, Stargate demonstrates acceptable performance in terms of time and memory consumption. With future integration of an exact arithmetic solver, we anticipate further performance improvements. This extension is highly feasible, as Stargate already includes support for exact arithmetic internally.

Overall, the current state of Stargate provides a robust foundation for further development and benchmarking of SPGs via their reduction to SSGs.

Future Work: As highlighted in the previous sections, a key limitation lies in the inability of PRISM-games to accurately handle very small values. Addressing this issue is essential for enabling Stargate to reach its full potential. A natural next step is to integrate an exact arithmetic solver, potentially based on rational or symbolic computation, to overcome floating-point rounding errors. A promising solver for this purpose is Storm [17] which currently only analyzes discrete- and continuous-time variants of Markov chains and Markov decision processes using exact arithmetic. With an extension for SGs, Stargate could integrate Storm to achieve potentially better results. Moreover, support for extracting strategies from SMGs and translating them back to the original SPGs would expand the applicability of Stargate toward synthesis and further analysis. This requires extending the current PRISM-games integration or replacing it with a more modern solver that supports strategy export.
In terms of usability, making the tool platform-independent and improving its performance, e.g., by parallelizing transformation steps, enhances its suitability for larger-scale applications. Finally, incorporating additional input formats and developing an automated benchmarking pipeline would foster more comprehensive evaluations and comparisons in future research.

Acknowledgements: I, Merlin Weise, would like to express my sincere gratitude to Alex Bork and Raphaël Berthon for their excellent supervision and the generous amount of time they dedicated to supporting my work.
I am particularly grateful to Raphaël for his guidance in introducing ε-optimality into the reduction—an approach he co-developed—and for recommending valuable literature that deepened my understanding of the subject.
I thank Alex for identifying the extended version of PRISM-games, which significantly improved the results obtained with Stargate.
Both supervisors jointly proposed the idea of using alternating vertices in the transformation from SSGs to SMGs, which became a central component of this work.
Additionally, I would like to thank Krishnendu Chatterjee and Arjun Radhakrishna for kindly answering my questions regarding GIST.

References

[1] Daniel Andersson and Peter Bro Miltersen. The complexity of solving stochastic games on graphs. In *Algorithms and Computation*, pages 112–121, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. The MIT Press, Cambridge, Massachusetts and London, England, 2008.

[3] Raphaël Berthon, Joost-Pieter Katoen, and Zihan Zhou. A direct reduction from stochastic parity games to simple stochastic games. To appear in Proceedings of CONCUR 2025, 2025.

[4] Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger, and Jean-François Raskin. Algorithms for omega-regular games with imperfect information. In *Computer Science Logic*, pages 287–302, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[5] Krishnendu Chatterjee and Nathanaël Fijalkow. A reduction from parity games to simple stochastic games. *Electronic Proceedings in Theoretical Computer Science*, 54:74–86, 2011.

[6] Krishnendu Chatterjee and Thomas A. Henzinger. Strategy improvement and randomized subexponential algorithms for stochastic parity games. In *STACS 2006*, volume 3884 of *Lecture Notes in Computer Science*, pages 512–523. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[7] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Arjun Radhakrishna. Gist: A solver for probabilistic games. In *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 665–669. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[8] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. Automatic verification of competitive stochastic systems. *Formal Methods in System Design*, 43(1):61–92, 2013.

[9] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. PRISM-games: A model checker for stochastic multi-player games. In *Tools and algorithms for the construction and analysis of systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 185–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[10] Anne Condon. On algorithms for simple stochastic games. In *Advances In Computational Complexity Theory*, 1990.

[11] Anne Condon. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992.

[12] Jerry Ding, Maryam Kamgarpour, Sean Summers, Alessandro Abate, John Lygeros, and Claire Tomlin. A stochastic games framework for verification and control of discrete time stochastic hybrid systems. *Automatica*, 49(9):2665–2674, 2013.

[13] Oliver Friedmann and Martin Lange. Solving parity games in practice. In *Automated technology for verification and analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 182–196. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[14] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, logics, and infinite games: a guide to current research*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[15] Serge Hadded and Benjamin Monmege. Interval iteration algorithm for MDPs and IMDPs. *Theoretical Computer Science*, 735:111–131, 2018.

[16] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Model-free reinforcement learning for stochastic parity games: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. *LIPIcs, Volume 171, CONCUR 2020*, 171, 2020.

[17] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. The probabilistic model checker Storm. *STTT*, 24(4):589–610, 2022.

[18] Jan Křetínský, Emanuel Ramneantu, Alexander Slivinskiy, and Maximilian Weininger. Comparison of algorithms for simple stochastic games. *Information and Computation*, 289:104885, 2022.

[19] Harold J. Kushner. The Gauss-Seidel numerical procedure for Markov stochastic games. *IEEE Transactions on Automatic Control*, 49(10):1779–1784, 2004.

[20] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV 2011*, volume 6806 of *LNCS*, pages 585–591, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[21] Marta Kwiatkowska, Gethin Norman, David Parker, and Gabriel Santos. PRISM-games 3.0: Stochastic game verification with concurrency, equilibria and time. In *CAV 2020*, volume 12225 of *LNCS*, pages 475–487, Berlin, Heidelberg, 2020. Springer Berlin Heidelberg.

[22] Donald A. Martin. The determinacy of Blackwell games. *Journal of Symbolic Logic*, 63(4):1565–1581, 1998.

[23] Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 255–264, 2006.

[24] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 2014.

[25] Lloyd S. Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953.

[26] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024.

[27] Ralf Wimmer, Alexander Kortus, Marc Herbstritt, and Bernd Becker. Probabilistic model checking and reliability of results. In *2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 1–6, 2008.

A Appendix

Notation for Appendix: In the following sections, we use the short notation:

- **# Vert.** for number of vertices
- **(V*X*)** for Version X
- **# S** for number of states in SMG
- **# T** for number of transitions in SMG
- **PCT** for property checking time in seconds
- **TT** for transformation time in seconds
- **Size** for SMG size in bytes
- **VIT** for property checking time with Value Iteration in seconds
- **PIT** for property checking time with Policy Iteration in seconds

A.1 Evaluation: SSG to SMG Transformation V1 vs. V2

A.1.1 Number of States and Transitions of SMG

Random SSGs

#Vert.	#S (V1)	#S (V2)	#T (V1)	#T (V2)
2	4	4	6	6
4	6	6	10	10
8	13	16	21	24
16	28	39	64	75
32	63	141	248	326
64	128	436	779	1087
128	256	1354	2665	3763
256	512	5301	10383	15172
512	1024	20496	40479	59951

Binary Tree SSGs

#Vert.	#S (V1)	#S (V2)	#T (V1)	#T (V2)
3	5	5	6	6
7	10	10	13	13
15	22	22	29	29
31	47	47	62	62
63	88	88	119	119
127	187	187	250	250
255	386	386	513	513
511	774	774	1029	1029
1023	1533	1533	2044	2044
2047	3089	3089	4112	4112
4095	6076	6076	8123	8123
8191	12287	12287	16382	16382
16383	24652	24652	32843	32843

Chain SSGs

#Vert.	#S (V1)	#S (V2)	#T (V1)	#T (V2)
2	4	4	5	5
4	6	7	9	10
8	11	15	18	22
16	22	30	37	45
32	47	57	78	88
64	94	126	157	189
128	187	252	314	379
256	374	503	629	758
512	760	1012	1271	1523
1024	1527	2028	2550	3051
2048	3057	4121	5104	6168
4096	6105	8115	10200	12210
8192	12297	16396	20488	24587

Complete Graph SSGs

#Vert.	#S (V1)	#S (V2)	#T (V1)	#T (V2)
2	2	2	4	4
4	7	10	19	22
8	15	50	71	106
16	32	130	272	370
32	64	514	1056	1506
64	128	2050	4160	6082
128	256	8290	16512	24546

Empty SSGs

#Vert.	#S (V1)	#S (V2)	#T (V1)	#T (V2)
2	1	1	1	1
4	1	1	1	1
8	1	1	1	1
16	1	1	1	1
32	1	1	1	1
64	1	1	1	1
128	1	1	1	1
256	1	1	1	1
512	1	1	1	1
1024	1	1	1	1
2048	1	1	1	1
4096	1	1	1	1
8192	1	1	1	1
16384	1	1	1	1
32768	1	1	1	1
65536	1	1	1	1
131072	1	1	1	1
262144	1	1	1	1

A.1.2 Transformation and Property Check Time

Random SSGs

#Vert.	TT (V1)	TT (V2)	PCT (V1)	PCT (V2)
2	0.0003	0.0002	11.9974	4.9804
4	0.0003	0.0002	5.0430	4.9533
8	0.0005	0.0007	5.2325	5.6877
16	0.0019	0.0011	5.4199	5.0822
32	0.0088	0.0045	5.0699	5.0346
64	0.0209	0.0257	5.6901	5.1995
128	0.0445	0.1432	5.4406	5.6637
256	0.1867	2.0161	7.7352	35.6251
512	2.7572	33.7240	23.7261	514.7915

Binary Tree SSGs

#Vert.	TT (V1)	TT (V2)	PCT (V1)	PCT (V2)
3	0.0002	0.0002	4.9609	4.8119
7	0.0003	0.0005	4.9682	4.8987
15	0.0018	0.0005	4.9246	4.9508
31	0.0014	0.0011	5.0237	4.8983
63	0.0028	0.0023	5.0251	4.9596
127	0.0033	0.0039	4.9321	4.9800
255	0.0091	0.0172	5.0411	5.0619
511	0.0263	0.0179	5.2155	5.0349
1023	0.0347	0.0458	5.2791	5.2550
2047	0.0527	0.1450	6.7418	6.7073
4095	0.1257	0.4200	17.3496	17.6814
8191	0.2896	1.5841	81.8009	82.0874
16383	0.7693	6.4379	329.7388	333.0548

Chain SSGs

#Vert.	TT (V1)	TT (V2)	PCT (V1)	PCT (V2)
2	0.0002	0.0002	5.0017	4.8714
4	0.0003	0.0003	4.8801	4.8930
8	0.0005	0.0006	4.9054	4.9433
16	0.0020	0.0010	4.9753	4.9475
32	0.0011	0.0014	4.9068	4.9244
64	0.0021	0.0034	4.9654	4.9388
128	0.0038	0.0079	4.9902	5.0165
256	0.0237	0.0138	5.0473	5.0156
512	0.0240	0.0365	5.0920	5.2161
1024	0.0388	0.1046	5.6272	5.9532
2048	0.0651	0.3895	8.1745	10.2078
4096	0.1729	1.3597	26.3291	40.6315
8192	0.4866	5.8550	126.5585	196.8695

Complete Graph SSGs

#Vert.	TT (V1)	TT (V2)	PCT (V1)	PCT (V2)
2	0.0002	0.0003	4.9551	4.9244
4	0.0006	0.0007	4.9324	4.8486
8	0.0022	0.0024	4.9183	4.9321
16	0.0047	0.0066	4.9995	5.0706
32	0.0275	0.0397	5.1822	5.4168
64	0.0706	0.3658	6.1673	11.0864
128	0.4071	5.3497	30.8948	144.2512

Empty SSGs

#Vert.	TT (V1)	TT (V2)	PCT (V1)	PCT (V2)
2	0.0005	0.0002	4.8917	4.8318
4	0.0002	0.0003	4.9396	4.9101
8	0.0003	0.0005	4.8036	4.8492
16	0.0016	0.0007	4.9212	4.8295
32	0.0010	0.0012	4.8699	4.8754
64	0.0016	0.0017	4.9049	4.9236
128	0.0080	0.0044	4.9748	4.9267
256	0.0078	0.0151	4.9401	5.0159
512	0.0198	0.0136	5.0104	5.0593
1024	0.0330	0.0234	5.1090	5.1087
2048	0.0421	0.0433	5.2602	5.1980
4096	0.0835	0.0985	5.5914	5.4639
8192	0.2057	0.2233	5.6843	5.6503
16384	0.4387	0.4491	6.1407	6.0990
32768	1.0167	1.0540	9.9706	6.8272
65536	2.2223	2.5012	11.6033	8.5900
131072	5.2590	6.9007	18.0287	12.7345

A.2 Evaluation: SSG to SMG Transformation V1 vs. V3

A.2.1 Number of States and Transitions of SMG

Random SSGs

#Vert.	#S (V1)	#S (V3)	#T (V1)	#T (V3)
2	4	6	6	8
4	5	9	8	12
8	13	21	20	28
16	29	70	70	111
32	64	232	244	412
64	128	720	767	1359
128	256	2499	2681	4924
256	512	9653	10370	19511

Binary Tree SSGs

#Vert.	#S (V1)	#S (V3)	#T (V1)	#T (V3)
3	4	5	5	6
7	9	13	12	16
15	22	29	29	36
31	47	61	62	76
63	102	125	133	156
127	188	253	251	316
255	380	509	507	636
512	755	1021	1010	1276
1023	1530	2045	2041	2556
2047	3080	4093	4103	5116
4095	6152	8189	8199	10236
8191	12341	16381	16436	20476
16383	24573	32765	32764	40956

Chain SSGs

#Vert.	#S (V1)	#S (V3)	#T (V1)	#T (V3)
2	4	4	5	5
4	6	10	9	13
8	12	22	19	29
16	26	46	41	61
32	48	94	79	125
64	98	190	161	253
128	200	382	327	509
256	364	766	619	1021
512	756	1534	1267	2045
1024	1531	3070	2554	4093
2048	3095	6142	5142	8189
4096	6170	12286	10265	16381
8192	12258	24574	20449	32765

Complete Graph SSGs

#Vert.	#S (V1)	#S (V3)	#T (V1)	#T (V3)
2	4	2	6	4
4	7	4	19	16
8	16	8	72	64
16	32	16	272	256
32	64	32	1056	1024
64	128	64	4160	4096
128	256	128	16512	16384

Empty SSGs

#Vert.	#S (V1)	#S (V3)	#T (V1)	#T (V3)
2	1	1	1	1
4	1	1	1	1
8	1	1	1	1
16	1	1	1	1
32	1	1	1	1
64	1	1	1	1
128	1	1	1	1
256	1	1	1	1
512	1	1	1	1
1024	1	1	1	1
2048	1	1	1	1
4096	1	1	1	1
8192	1	1	1	1
16384	1	1	1	1
32768	1	1	1	1
65536	1	1	1	1

A.2.2 Transformation and Property Check Time

Random SSGs

#Vert.	TT (V1)	TT (V3)	PCT (V1)	PCT (V3)
2	0.0003	0.0001	10.4862	5.1674
4	0.0002	0.0001	5.1575	5.0938
8	0.0004	0.0001	5.1460	5.0885
16	0.0010	0.0003	5.3087	5.2001
32	0.0031	0.0013	5.3251	5.2021
64	0.0210	0.0076	5.3810	5.3208
128	0.0470	0.0720	5.6240	6.3933
256	0.1885	1.1657	7.8484	61.3441

Binary Tree SSGs

#Vert.	TT (V1)	TT (V3)	PCT (V1)	PCT (V3)
3	0.0002	0.0001	6.0681	5.0373
7	0.0003	0.0001	5.0851	4.9973
15	0.0008	0.0001	5.0291	5.0664
31	0.0030	0.0002	5.4208	5.0528
63	0.0031	0.0007	5.0837	4.9665
127	0.0034	0.0015	5.0458	5.0166
255	0.0068	0.0036	5.1559	5.2866
511	0.0254	0.0100	5.3495	5.2514
1023	0.0427	0.0338	5.4526	5.3803
2047	0.0561	0.1214	6.9585	6.7662
4095	0.1290	0.4818	19.4486	17.1259
8191	0.3198	2.1977	87.6350	92.7456
16383	0.7837	10.3559	336.0399	377.9834

Chain SSGs

#Vert.	TT (V1)	TT (V3)	PCT (V1)	PCT (V3)
2	0.0003	0.0001	5.0568	4.8372
4	0.0002	0.0001	4.9806	4.8881
8	0.0004	0.0001	4.9574	4.9057
16	0.0010	0.0002	4.9503	4.8807
32	0.0020	0.0003	4.9048	4.9740
64	0.0040	0.0004	4.9223	4.9367
128	0.0040	0.0012	4.9624	5.0247
256	0.0204	0.0034	5.0773	5.0628
512	0.0244	0.0097	5.1641	5.0520
1024	0.0396	0.0294	5.8479	6.5246
2048	0.0633	0.1131	8.5443	12.218
4096	0.1617	0.4529	31.9953	44.0424
8192	0.4626	2.0171	132.8163	288.3853

Complete Graph SSGs

#Vert.	TT (V1)	TT (V3)	PCT (V1)	PCT (V3)
2	0.0003	0.0001	5.2099	4.9002
4	0.0014	0.0002	5.1428	5.0847
8	0.0030	0.0007	4.8888	4.8911
16	0.0059	0.0043	5.0294	4.9569
32	0.0267	0.0510	5.2522	5.2379
64	0.0724	0.7309	6.3112	6.4000
128	0.4169	14.0531	32.0438	30.9459

Empty SSGs

#Vert.	TT (V1)	TT (V3)	PCT (V1)	PCT (V3)
2	0.0001	0.0001	4.9500	4.852
4	0.0002	0.0001	4.9252	4.8734
8	0.0003	0.0001	4.9036	4.8910
16	0.0005	0.0001	4.8803	4.9539
32	0.0018	0.0002	4.9143	4.8748
64	0.0014	0.0007	4.9001	4.9076
128	0.0027	0.0015	5.0817	4.8724
256	0.0051	0.0039	5.0745	4.9650
512	0.0206	0.0131	4.9811	5.0768
1024	0.0294	0.0478	5.1164	5.2143
2048	0.0413	0.1883	5.2009	5.3995
4096	0.0845	0.7491	5.5438	5.5784
8192	0.1908	2.9803	5.6897	5.8542
16384	0.4344	13.9155	6.1086	6.9193
32768	1.0488	75.7272	10.2298	13.0437
65536	2.4740	326.9754	13.6839	22.4505

A.3 Evaluation: Scalability

Vertices: 2

Vertices: 2, Number of Outgoing Transitions: 1, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0004	904	6.0479	2.9525
10^{-6}	0.0004	912	2.9439	2.8476
10^{-2}	0.0007	912	2.8588	2.8233
10^{-1}	0.0004	912	2.9680	2.8530

Vertices: 2, Number of Outgoing Transitions: 2, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0005	1104	2.8703	2.9018
10^{-6}	0.0006	1104	2.9563	3.2186
10^{-2}	0.0006	1104	2.8650	2.9151
10^{-1}	0.0005	1104	3.0960	2.8727

Vertices: 3

Vertices: 3, Number of Outgoing Transitions: 1, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0004	1120	3.0175	2.8273
10^{-6}	0.0005	1136	2.9485	2.8651
10^{-2}	0.0005	1136	2.8540	2.8644
10^{-1}	0.0004	1136	2.9159	2.8211

Vertices: 3, Number of Outgoing Transitions: 2, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0005	1608	2.8855	2.9129
10^{-6}	0.0005	1624	2.8897	2.8281
10^{-2}	0.0005	1624	2.9114	2.8630
10^{-1}	0.0005	1624	2.9051	2.8648

Vertices: 3, Number of Outgoing Transitions: 3, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0005	1576	2.8811	2.8242
10^{-6}	0.0005	1608	2.9100	2.8774
10^{-2}	0.0006	1608	2.9651	2.8382
10^{-1}	0.0005	1608	2.8857	2.9494

Vertices: 8

Vertices: 8, Number of Outgoing Transitions: 1, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0008	2568	2.8415	3.2939
10^{-6}	0.0008	2632	3.4497	3.2592
10^{-2}	0.0007	2696	3.1715	2.9479
10^{-1}	0.0007	2688	2.9874	2.9328

Vertices: 8, Number of Outgoing Transitions: 1, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0007	2512	2.8645	3.1047
10^{-6}	0.0007	2520	2.9098	3.0324
10^{-2}	0.0008	2552	3.6128	3.1788
10^{-1}	0.0008	2624	3.2619	2.9934

Vertices: 8, Number of Outgoing Transitions: 1, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0007	2584	2.8053	3.1073
10^{-6}	0.0008	2584	3.6989	3.2229
10^{-2}	0.0008	2608	3.5963	3.1090
10^{-1}	0.0007	2648	3.1675	2.9031

Vertices: 8, Number of Outgoing Transitions: 2, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0009	3088	2.8896	2.8733
10^{-6}	0.0009	3136	2.8708	2.8623
10^{-2}	0.0009	3216	2.9069	2.8833
10^{-1}	0.0009	3208	2.9143	2.8894

Vertices: 8, Number of Outgoing Transitions: 2, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0009	3432	2.8459	3.3337
10^{-6}	0.0010	3440	4.4129	3.9849
10^{-2}	0.0009	3456	4.2545	3.7866
10^{-1}	0.0011	3520	3.3117	3.5238

Vertices: 8, Number of Outgoing Transitions: 2, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0009	3424	2.9019	2.8471
10^{-6}	0.0009	3440	2.8334	2.8469
10^{-2}	0.0010	3488	2.9009	3.1473
10^{-1}	0.0010	3496	2.8550	2.8986

Vertices: 8, Number of Outgoing Transitions: 4, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0013	4768	2.8532	3.1872
10^{-6}	0.0012	4816	4.0431	3.9592
10^{-2}	0.0014	4888	3.8137	4.7400
10^{-1}	0.0013	4888	3.1729	3.1244

Vertices: 8, Number of Outgoing Transitions: 4, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0012	4728	2.8813	2.8889
10^{-6}	0.0020	4744	2.9004	2.9570
10^{-2}	0.0019	4776	2.9456	2.9751
10^{-1}	0.0013	4800	2.9450	2.9187

Vertices: 8, Number of Outgoing Transitions: 4, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0012	4840	2.9884	2.8772
10^{-6}	0.0014	4856	2.9639	2.9330
10^{-2}	0.0013	4904	2.8996	2.9069
10^{-1}	0.0013	4928	3.0052	2.8951

Vertices: 8, Number of Outgoing Transitions: 8, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0021	7408	2.9638	2.9152
10^{-6}	0.0038	7512	5.0930	2.8908
10^{-2}	0.0018	7536	3.9363	2.8855
10^{-1}	0.0019	7528	2.9352	2.9112

Vertices: 8, Number of Outgoing Transitions: 8, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0115	7560	2.9001	2.9313
10^{-6}	0.0019	7608	5.2647	7.5184
10^{-2}	0.0019	7656	4.8200	600.0000
10^{-1}	0.0022	7680	3.6099	600.0000

Vertices: 8, Number of Outgoing Transitions: 8, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0021	7424	3.0062	3.7788
10^{-6}	0.0077	7432	3.7061	2.9772
10^{-2}	0.0056	7448	3.1906	3.7600
10^{-1}	0.0056	7472	3.6637	3.0591

Vertices: 15

Vertices: 15, Number of Outgoing Transitions: 1, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0032	4320	3.0387	3.5773
10^{-6}	0.0051	4560	4.4873	3.5774
10^{-2}	0.0033	4664	5.1202	3.5874
10^{-1}	0.0023	4672	4.1110	4.2959

Vertices: 15, Number of Outgoing Transitions: 1, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0021	4240	4.0674	2.9238
10^{-6}	0.0021	4496	5.1277	4.8667
10^{-2}	0.0015	4528	4.8910	3.8958
10^{-1}	0.0024	4536	5.7772	3.8835

Vertices: 15, Number of Outgoing Transitions: 1, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0019	3984	3.1835	3.6815
10^{-6}	0.0022	4248	3.4270	3.7639
10^{-2}	0.0017	4264	5.3771	3.7149
10^{-1}	0.0014	4272	4.8604	4.2282

Vertices: 15, Number of Outgoing Transitions: 2, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0071	5384	3.8066	3.0137
10^{-6}	0.0023	5632	5.9805	8.2369
10^{-2}	0.0046	5768	5.9044	8.4636
10^{-1}	0.0042	5768	5.9281	6.8996

Vertices: 15, Number of Outgoing Transitions: 2, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0044	5664	3.0976	3.3035
10^{-6}	0.0031	5912	7.4655	8.1807
10^{-2}	0.0041	5944	6.7507	7.5760
10^{-1}	0.0029	5952	6.0862	8.1229

Vertices: 15, Number of Outgoing Transitions: 2, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0044	5576	2.9732	2.9958
10^{-6}	0.0040	5840	5.1645	600.0000
10^{-2}	0.0016	5856	5.1667	600.0000
10^{-1}	0.0018	5864	4.8738	600.0000

Vertices: 15, Number of Outgoing Transitions: 3, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0022	7080	3.1151	3.0893
10^{-6}	0.0020	7312	6.0646	5.9554
10^{-2}	0.0017	7384	6.1043	5.2888
10^{-1}	0.0032	7392	6.1958	4.6069

Vertices: 15, Number of Outgoing Transitions: 3, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0017	7048	3.6010	3.6152
10^{-6}	0.0020	7304	5.1880	5.6859
10^{-2}	0.0019	7376	5.3710	5.6727
10^{-1}	0.0051	7384	6.5728	5.0863

Vertices: 15, Number of Outgoing Transitions: 3, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0016	6840	3.4377	3.5927
10^{-6}	0.0035	7112	7.3400	600.0000
10^{-2}	0.0020	7120	6.9096	600.0000
10^{-1}	0.0031	7128	6.5596	600.0000

Vertices: 15, Number of Outgoing Transitions: 8, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0036	13472	3.0652	2.9334
10^{-6}	0.0040	13712	6.6589	8.0129
10^{-2}	0.0035	13816	6.5726	6.2354
10^{-1}	0.0035	13824	6.4359	5.8869

Vertices: 15, Number of Outgoing Transitions: 8, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0032	13376	2.9379	2.9748
10^{-6}	0.0033	13632	3.7212	5.0239
10^{-2}	0.0034	13688	3.1055	4.5173
10^{-1}	0.0040	13704	4.4888	3.8869

Vertices: 15, Number of Outgoing Transitions: 8, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0032	13352	3.0749	3.7887
10^{-6}	0.0038	13608	3.7166	3.0951
10^{-2}	0.0044	13648	3.0039	3.6814
10^{-1}	0.0057	13656	3.3614	2.8928

Vertices: 15, Number of Outgoing Transitions: 15, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0077	22128	2.9749	3.7687
10^{-6}	0.0063	22368	9.4604	5.5947
10^{-2}	0.0056	22488	8.9604	5.1425
10^{-1}	0.0054	22496	9.8695	4.7220

Vertices: 15, Number of Outgoing Transitions: 15, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0108	22368	3.9109	2.9927
10^{-6}	0.0053	22640	3.0539	3.4374
10^{-2}	0.0066	22696	3.8451	3.0761
10^{-1}	0.0090	22704	3.4204	3.7634

Vertices: 15, Number of Outgoing Transitions: 15, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0076	22272	3.5544	3.1134
10^{-6}	0.0053	22528	3.0136	3.6332
10^{-2}	0.0054	22560	3.4782	3.0350
10^{-1}	0.0084	22568	3.0622	3.9832

Vertices: 32

Vertices: 32, Number of Outgoing Transitions: 2, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0028	11504	3.4384	2.9940
10^{-6}	0.0058	12112	2.9800	14.3140
10^{-2}	0.0067	12360	3.0597	14.1517
10^{-1}	0.0033	12344	8.3005	13.7993

Vertices: 32, Number of Outgoing Transitions: 2, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0030	11416	3.0609	3.6875
10^{-6}	0.0029	12008	3.5589	3.0364
10^{-2}	0.0030	12192	2.9566	3.6272
10^{-1}	0.0039	12184	3.4115	3.0822

Vertices: 32, Number of Outgoing Transitions: 2, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0029	11232	2.9824	3.4658
10^{-6}	0.0034	11840	3.5362	11.4921
10^{-2}	0.0036	11888	3.1010	12.3561
10^{-1}	0.0029	11864	10.0198	12.2557

Vertices: 32, Number of Outgoing Transitions: 2, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0037	11208	4.0390	3.4937
10^{-6}	0.0030	11712	3.0223	5.5554
10^{-2}	0.0042	11840	3.6671	12.9186
10^{-1}	0.0029	11824	9.8534	12.6795

Vertices: 32, Number of Outgoing Transitions: 4, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0042	17216	3.0845	3.3822
10^{-6}	0.0068	17824	3.1881	17.4126
10^{-2}	0.0063	18136	3.0370	17.7640
10^{-1}	0.0074	18128	9.9719	17.7304

Vertices: 32, Number of Outgoing Transitions: 4, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0042	17216	3.0320	3.0592
10^{-6}	0.0052	17808	3.6262	3.5618
10^{-2}	0.0055	17920	3.3011	2.9856
10^{-1}	0.0046	17912	11.2583	7.0042

Vertices: 32, Number of Outgoing Transitions: 4, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0042	17072	3.1497	2.9377
10^{-6}	0.0049	17680	2.9311	7.8275
10^{-2}	0.0045	17712	2.9117	7.1874
10^{-1}	0.0045	17696	3.5541	6.1973

Vertices: 32, Number of Outgoing Transitions: 4, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0062	17176	3.5827	3.5704
10^{-6}	0.0124	17664	2.9364	3.2612
10^{-2}	0.0108	17800	3.8184	3.5642
10^{-1}	0.0125	17792	3.0168	2.9613

Vertices: 32, Number of Outgoing Transitions: 7, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0077	25304	4.0197	3.2411
10^{-6}	0.0074	25912	3.1218	12.1176
10^{-2}	0.0187	26184	3.0079	14.5634
10^{-1}	0.0187	26168	29.0845	12.2354

Vertices: 32, Number of Outgoing Transitions: 7, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0125	25136	3.1036	3.1678
10^{-6}	0.0134	25728	3.7529	3.0436
10^{-2}	0.0100	25808	2.9532	3.1972
10^{-1}	0.0137	25792	3.4620	2.9934

Vertices: 32, Number of Outgoing Transitions: 7, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0158	25128	2.9969	3.0477
10^{-6}	0.0130	25728	4.1671	3.2110
10^{-2}	0.0180	25792	2.9237	3.3119
10^{-1}	0.0185	25768	14.9133	3.0756

Vertices: 32, Number of Outgoing Transitions: 7, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0173	25096	3.0979	2.9812
10^{-6}	0.0181	25704	3.2907	3.9061
10^{-2}	0.0137	25768	2.9899	2.9750
10^{-1}	0.0155	25760	3.3572	3.7349

Vertices: 32, Number of Outgoing Transitions: 16, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0165	49456	3.0626	3.1395
10^{-6}	0.0467	50064	3.7052	3.6375
10^{-2}	0.0299	50304	3.0863	3.0224
10^{-1}	0.0148	50288	3.4568	3.7089

Vertices: 32, Number of Outgoing Transitions: 16, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0292	49592	3.1446	3.1270
10^{-6}	0.0147	50184	3.7990	3.4991
10^{-2}	0.0310	50336	3.0301	2.9640
10^{-1}	0.0374	50320	3.6723	3.4822

Vertices: 32, Number of Outgoing Transitions: 16, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0361	50536	3.0146	3.0276
10^{-6}	0.0238	51144	4.2915	3.4581
10^{-2}	0.0266	51176	3.1242	3.0059
10^{-1}	0.0357	51160	3.9363	3.4997

Vertices: 32, Number of Outgoing Transitions: 16, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0379	50104	3.0033	3.0373
10^{-6}	0.0311	50648	3.8537	3.1169
10^{-2}	0.0148	50736	3.1292	3.2411
10^{-1}	0.0490	50728	4.4189	3.2162

Vertices: 32, Number of Outgoing Transitions: 32, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0371	93480	3.1846	3.0825
10^{-6}	0.0678	94088	3.9574	3.2585
10^{-2}	0.0431	94280	3.1548	3.4075
10^{-1}	0.0575	94264	4.0777	3.0825

Vertices: 32, Number of Outgoing Transitions: 32, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0278	93232	3.1560	3.7416
10^{-6}	0.0686	93824	4.0259	3.1435
10^{-2}	0.0418	93928	3.1604	4.2764
10^{-1}	0.0604	93904	3.6631	3.0866

Vertices: 32, Number of Outgoing Transitions: 32, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0756	93528	3.2560	3.9630
10^{-6}	0.0371	94136	3.2433	3.1963
10^{-2}	0.0654	94176	3.7868	4.3161
10^{-1}	0.0696	94144	3.2900	3.0675

Vertices: 32, Number of Outgoing Transitions: 32, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0302	93288	4.0100	3.1470
10^{-6}	0.0465	93872	3.2385	3.1680
10^{-2}	0.0444	93912	4.4200	3.1216
10^{-1}	0.0456	93904	3.2522	3.1130

Vertices: 63

Vertices: 63, Number of Outgoing Transitions: 4, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0161	33296	3.8206	3.0681
10^{-6}	0.0125	34408	3.1259	8.9445
10^{-2}	0.0288	34448	3.0141	9.2081
10^{-1}	0.0187	34472	3.0641	9.1871

Vertices: 63, Number of Outgoing Transitions: 4, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0280	33240	3.4772	2.9525
10^{-6}	0.0268	34344	3.1193	3.6329
10^{-2}	0.0102	34384	3.6811	3.0528
10^{-1}	0.0222	34392	3.0676	3.9200

Vertices: 63, Number of Outgoing Transitions: 4, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0198	33176	3.4361	3.0116
10^{-6}	0.0106	34328	3.1910	3.8863
10^{-2}	0.0130	34344	4.2751	3.4255
10^{-1}	0.0086	34360	3.0077	3.0346

Vertices: 63, Number of Outgoing Transitions: 4, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0176	33280	3.8139	2.9308
10^{-6}	0.0091	33712	3.0056	2.9869
10^{-2}	0.0111	33784	4.0431	3.2236
10^{-1}	0.0087	33808	3.1158	3.0042

Vertices: 63, Number of Outgoing Transitions: 7, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0141	48800	3.6922	3.1057
10^{-6}	0.0128	49912	3.2195	3.1885
10^{-2}	0.0305	49944	3.8261	3.0549
10^{-1}	0.0131	49968	3.0307	3.0190

Vertices: 63, Number of Outgoing Transitions: 7, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0155	49504	3.9584	3.1345
10^{-6}	0.0141	50616	3.0574	14.8631
10^{-2}	0.0246	50648	3.2030	15.1097
10^{-1}	0.0129	50656	3.7893	15.1162

Vertices: 63, Number of Outgoing Transitions: 7, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0131	49368	3.1023	3.0722
10^{-6}	0.0137	50528	3.4689	3.8625
10^{-2}	0.0156	50536	3.0981	3.1259
10^{-1}	0.0148	50560	4.0301	3.5054

Vertices: 63, Number of Outgoing Transitions: 7, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0191	49104	3.0570	3.0308
10^{-6}	0.0132	49640	3.8988	3.5669
10^{-2}	0.0172	49712	3.0296	3.0327
10^{-1}	0.0139	49736	3.9105	3.2934

Vertices: 63, Number of Outgoing Transitions: 13, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0221	81368	3.1660	3.2486
10^{-6}	0.0414	82472	3.9292	3.2326
10^{-2}	0.0271	82504	3.1113	3.1421
10^{-1}	0.0228	82536	3.9866	3.0990

Vertices: 63, Number of Outgoing Transitions: 13, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0214	81936	3.1161	3.0612
10^{-6}	0.0228	83048	4.0108	3.1298
10^{-2}	0.0221	83072	3.1673	3.3727
10^{-1}	0.0417	83080	4.4176	3.1058

Vertices: 63, Number of Outgoing Transitions: 13, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0326	82344	3.0368	3.5933
10^{-6}	0.0235	83504	3.9777	3.2141
10^{-2}	0.0234	83504	3.1198	4.3243
10^{-1}	0.0410	83528	3.5336	3.1406

Vertices: 63, Number of Outgoing Transitions: 13, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0309	81856	3.1475	4.1512
10^{-6}	0.0351	82368	3.2911	3.0559
10^{-2}	0.0381	82472	3.1742	3.9775
10^{-1}	0.0308	82504	3.1753	3.0856

Vertices: 63, Number of Outgoing Transitions: 32, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0577	184696	3.4440	3.7533
10^{-6}	0.0640	185808	3.2334	3.2846
10^{-2}	0.0517	185840	3.4686	3.7429
10^{-1}	0.0511	185872	3.1468	3.1891

Vertices: 63, Number of Outgoing Transitions: 32, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0727	185416	3.6943	3.8577
10^{-6}	0.0518	186536	3.2351	3.1253
10^{-2}	0.0756	186568	4.0811	3.5993
10^{-1}	0.0518	186568	3.1948	3.3852

Vertices: 63, Number of Outgoing Transitions: 32, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0779	185928	3.6501	3.1371
10^{-6}	0.0514	187080	3.1841	3.5222
10^{-2}	0.0714	187088	4.0101	3.2064
10^{-1}	0.0638	187112	3.1728	3.7004

Vertices: 63, Number of Outgoing Transitions: 32, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.1166	185384	4.1172	3.1189
10^{-6}	0.0665	185888	3.1913	4.0615
10^{-2}	0.0672	185944	3.4487	3.1728
10^{-1}	0.0689	185976	3.2540	4.2140

Vertices: 63, Number of Outgoing Transitions: 63, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.1883	353896	3.3075	3.3345
10^{-6}	0.1325	355000	4.1299	3.9720
10^{-2}	0.1072	355024	3.3303	3.5182
10^{-1}	0.2209	355064	4.7338	3.3573

Vertices: 63, Number of Outgoing Transitions: 63, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.1256	353112	3.2747	4.3555
10^{-6}	0.1545	354224	3.8582	3.3812
10^{-2}	0.1620	354256	3.9709	4.2013
10^{-1}	0.1312	354264	3.4035	3.3048

Vertices: 63, Number of Outgoing Transitions: 63, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.1313	353088	4.2472	3.4940
10^{-6}	0.1517	354240	3.5907	3.6145
10^{-2}	0.1596	354248	4.3845	3.3692
10^{-1}	0.1282	354264	3.3716	4.2213

Vertices: 63, Number of Outgoing Transitions: 63, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.1879	352968	3.9097	3.2500
10^{-6}	0.1458	353496	3.4856	4.0148
10^{-2}	0.1116	353584	3.5846	3.4817
10^{-1}	0.1292	353616	4.1558	3.8619

Vertices: 128

Vertices: 128, Number of Outgoing Transitions: 7, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0317	100704	3.0495	3.1655
10^{-6}	0.0405	103072	4.2747	3.4286
10^{-2}	0.0298	103016	3.3004	3.2573
10^{-1}	0.0548	103072	3.7763	3.1506

Vertices: 128, Number of Outgoing Transitions: 7, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0301	101576	3.1302	3.1606
10^{-6}	0.0435	104048	4.1649	3.2413
10^{-2}	0.0425	103968	3.0471	3.9190
10^{-1}	0.0304	104016	3.8048	3.2295

Vertices: 128, Number of Outgoing Transitions: 7, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0389	100976	3.4031	4.1625
10^{-6}	0.0350	103024	3.5784	3.0025
10^{-2}	0.0337	103232	3.1440	4.1110
10^{-1}	0.0404	103256	3.5213	3.0664

Vertices: 128, Number of Outgoing Transitions: 7, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.0751	101584	3.0853	4.0737
10^{-6}	0.0312	102112	3.1473	3.2386
10^{-2}	0.0335	102128	3.7075	3.8609
10^{-1}	0.0306	102128	3.3178	3.2190

Vertices: 128, Number of Outgoing Transitions: 13, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.0651	167824	4.0248	3.6910
10^{-6}	0.0504	170192	3.2898	3.2929
10^{-2}	0.0774	170128	4.7053	3.1485
10^{-1}	0.0502	170192	3.1860	3.7215

Vertices: 128, Number of Outgoing Transitions: 13, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.0577	168616	3.8492	3.1499
10^{-6}	0.0636	171088	3.2402	4.0730
10^{-2}	0.0779	171024	3.6087	3.1591
10^{-1}	0.0542	171056	3.9415	3.9593

Vertices: 128, Number of Outgoing Transitions: 13, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.0742	168312	3.4732	3.2682
10^{-6}	0.0517	170240	4.4358	3.8032
10^{-2}	0.0659	170368	3.2233	3.4829
10^{-1}	0.0792	170384	4.2240	3.1971

Vertices: 128, Number of Outgoing Transitions: 13, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.1221	168424	3.3121	4.2947
10^{-6}	0.1036	168872	3.2409	3.2238
10^{-2}	0.0498	168920	3.1805	4.3987
10^{-1}	0.0686	168920	3.1383	3.1181

Vertices: 128, Number of Outgoing Transitions: 26, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.1152	315176	3.8454	3.8943
10^{-6}	0.1233	317544	3.3127	3.3910
10^{-2}	0.1235	317480	4.7396	3.4407
10^{-1}	0.1369	317544	3.3327	3.7095

Vertices: 128, Number of Outgoing Transitions: 26, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.1579	313120	4.5554	3.3017
10^{-6}	0.1521	315584	3.5197	4.7955
10^{-2}	0.1652	315520	3.6086	3.3612
10^{-1}	0.1337	315560	4.4343	3.9704

Vertices: 128, Number of Outgoing Transitions: 26, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.1460	313304	3.2277	3.4998
10^{-6}	0.1272	315104	4.9663	3.2302
10^{-2}	0.1229	315288	3.5012	4.3476
10^{-1}	0.1619	315296	3.7066	3.4302

Vertices: 128, Number of Outgoing Transitions: 26, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.1953	313816	3.8049	4.1857
10^{-6}	0.1307	314152	3.4315	3.4173
10^{-2}	0.1092	314200	4.3616	3.5544
10^{-1}	0.1381	314200	3.3577	3.5411

Vertices: 128, Number of Outgoing Transitions: 64, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.4107	738272	5.2145	4.3695
10^{-6}	0.3507	740632	5.1514	4.8835
10^{-2}	0.3435	740560	4.4077	5.7333
10^{-1}	0.4816	740632	4.5361	4.7500

Vertices: 128, Number of Outgoing Transitions: 64, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.4122	738672	5.2146	4.6733
10^{-6}	0.3762	741136	5.2946	5.5617
10^{-2}	0.3869	741072	4.6357	4.5360
10^{-1}	0.3409	741104	4.4501	5.4496

Vertices: 128, Number of Outgoing Transitions: 64, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.4252	739376	4.4794	5.1092
10^{-6}	0.3736	741344	5.0397	4.3073
10^{-2}	0.3697	741456	5.5862	4.5079
10^{-1}	0.3764	741480	4.4188	5.3303

Vertices: 128, Number of Outgoing Transitions: 64, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.4359	739792	4.4548	4.7344
10^{-6}	0.5041	740424	5.3147	4.5189
10^{-2}	0.3492	740456	5.2615	4.6400
10^{-1}	0.3391	740456	4.6063	5.8844

Vertices: 128, Number of Outgoing Transitions: 128, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.7403	1463832	5.9241	6.8308
10^{-6}	1.0055	1466200	5.9497	6.7528
10^{-2}	0.9744	1466136	6.0689	6.9519
10^{-1}	0.8506	1466200	5.9605	6.7070

Vertices: 128, Number of Outgoing Transitions: 128, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	1.0182	1464480	5.9835	6.7652
10^{-6}	1.2801	1466936	6.2059	6.4203
10^{-2}	0.9744	1466880	6.3967	6.3029
10^{-1}	0.8904	1466912	6.2287	6.4490

Vertices: 128, Number of Outgoing Transitions: 128, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	1.0016	1463624	6.0068	6.6889
10^{-6}	0.8192	1465448	6.2106	6.8426
10^{-2}	0.9187	1465600	6.2719	7.1331
10^{-1}	0.9890	1465608	6.3622	7.2323

Vertices: 128, Number of Outgoing Transitions: 128, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.9046	1462336	6.3982	6.6826
10^{-6}	0.9423	1462928	6.3230	6.5491
10^{-2}	0.8293	1462992	6.1502	6.9144
10^{-1}	0.9244	1463000	6.5711	6.7614

Vertices: 255

Vertices: 255, Number of Outgoing Transitions: 13, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.1800	339016	3.4250	4.3305
10^{-6}	0.1689	343984	3.9279	3.6525
10^{-2}	0.1700	343856	4.4750	4.2687
10^{-1}	0.1345	343984	3.6559	3.9095

Vertices: 255, Number of Outgoing Transitions: 13, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.2441	338704	4.5376	3.6072
10^{-6}	0.1459	342544	3.5586	4.8329
10^{-2}	0.1655	342832	3.7348	3.6676
10^{-1}	0.1564	343192	4.5752	4.0288

Vertices: 255, Number of Outgoing Transitions: 13, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.1921	338792	3.4604	4.2403
10^{-6}	0.2630	340720	5.0572	3.7174
10^{-2}	0.1343	340824	4.3790	4.4347
10^{-1}	0.1615	340984	3.7256	3.7457

Vertices: 255, Number of Outgoing Transitions: 13, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.5900	338400	4.7461	3.5915
10^{-6}	0.1541	338736	3.5730	4.9140
10^{-2}	0.2483	338768	3.9031	3.6411
10^{-1}	0.1777	338816	5.0524	3.9547

Vertices: 255, Number of Outgoing Transitions: 26, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.3510	629368	4.6845	5.8345
10^{-6}	0.3143	634344	5.0122	5.5950
10^{-2}	0.5645	634216	5.4474	4.8131
10^{-1}	0.3577	634344	6.5416	5.1672

Vertices: 255, Number of Outgoing Transitions: 26, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.3262	630464	6.0264	4.9753
10^{-6}	0.3768	634032	4.8972	5.9580
10^{-2}	0.3115	634432	4.9217	5.5870
10^{-1}	0.3539	634856	5.3220	4.9761

Vertices: 255, Number of Outgoing Transitions: 26, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.5850	632032	6.1380	4.6894
10^{-6}	0.3442	634112	5.6769	4.9375
10^{-2}	0.3171	634248	4.8878	5.5338
10^{-1}	0.3344	634432	4.8390	6.0206

Vertices: 255, Number of Outgoing Transitions: 26, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	0.8522	630296	5.2525	5.0252
10^{-6}	0.3550	630728	6.0193	4.8949
10^{-2}	0.3981	630768	5.8882	5.1135
10^{-1}	0.3493	630816	5.4837	6.2290

Vertices: 255, Number of Outgoing Transitions: 51, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	0.7670	1196576	7.5380	8.0167
10^{-6}	0.9548	1201544	7.2084	8.5053
10^{-2}	0.7732	1201424	6.9390	6.9680
10^{-1}	0.8534	1201544	7.7914	7.7492

Vertices: 255, Number of Outgoing Transitions: 51, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	0.8002	1195544	6.8435	6.9704
10^{-6}	0.8738	1199304	7.1356	7.0067
10^{-2}	0.9085	1199616	7.0381	7.2029
10^{-1}	0.8923	1200008	6.7779	7.4713

Vertices: 255, Number of Outgoing Transitions: 51, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	0.7682	1194032	6.7417	6.7110
10^{-6}	0.8248	1196000	6.8448	8.1012
10^{-2}	0.7780	1196160	6.9734	7.7296
10^{-1}	0.7356	1196360	7.1658	7.2287

Vertices: 255, Number of Outgoing Transitions: 51, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	1.1125	1195680	7.5057	6.7822
10^{-6}	0.8316	1196112	8.1688	7.2348
10^{-2}	0.9272	1196120	8.2384	7.0317
10^{-1}	0.8414	1196144	7.4065	7.2130

Vertices: 255, Number of Outgoing Transitions: 128, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	2.2988	2944664	12.5985	12.6244
10^{-6}	2.6060	2949640	12.7473	12.6230
10^{-2}	2.7039	2949504	12.8961	12.5623
10^{-1}	2.6234	2949640	13.1883	12.3858

Vertices: 255, Number of Outgoing Transitions: 128, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	2.4850	2945616	12.7389	12.4598
10^{-6}	2.9755	2949432	13.5124	12.6982
10^{-2}	2.9724	2949728	13.2980	12.9245
10^{-1}	2.8345	2950096	12.9961	12.4133

Vertices: 255, Number of Outgoing Transitions: 128, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	2.8165	2945976	13.2264	12.4186
10^{-6}	2.6699	2947808	13.9573	12.9007
10^{-2}	2.8958	2948000	14.1381	12.5737
10^{-1}	3.2360	2948216	12.8005	12.4855

Vertices: 255, Number of Outgoing Transitions: 128, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	2.9445	2943680	12.7579	12.5304
10^{-6}	2.7057	2944288	13.1462	12.4582
10^{-2}	2.8604	2944320	13.3922	12.9330
10^{-1}	2.5333	2944368	13.4404	12.6194

Vertices: 255, Number of Outgoing Transitions: 255, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	5.0844	5860376	21.4616	22.2713
10^{-6}	5.8300	5865352	21.7202	21.5758
10^{-2}	5.2753	5865216	21.6528	21.3624
10^{-1}	5.8408	5865352	21.5533	21.9745

Vertices: 255, Number of Outgoing Transitions: 255, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	5.5157	5873728	21.3494	22.0529
10^{-6}	5.6546	5877704	21.5820	22.7292
10^{-2}	6.5160	5877936	22.8613	21.9340
10^{-1}	5.6548	5878280	22.0158	21.9849

Vertices: 255, Number of Outgoing Transitions: 255, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	6.1113	5872944	22.2317	21.4471
10^{-6}	5.0302	5874960	22.2362	22.5932
10^{-2}	5.9196	5875144	22.3581	22.6800
10^{-1}	5.5298	5875360	22.0175	21.7071

Vertices: 255, Number of Outgoing Transitions: 255, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	5.7718	5870904	21.4857	21.3629
10^{-6}	5.5428	5871416	21.7448	21.5305
10^{-2}	5.8539	5871448	21.8139	21.2747
10^{-1}	5.4445	5871488	21.7349	21.8782

Vertices: 512

Vertices: 512, Number of Outgoing Transitions: 26, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	1.2037	1279296	10.0890	10.9049
10^{-6}	1.1371	1285224	10.1927	10.6845
10^{-2}	1.1535	1287256	10.5597	10.0554
10^{-1}	1.1504	1288016	10.3359	10.2535

Vertices: 512, Number of Outgoing Transitions: 26, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	1.0603	1279496	10.1108	10.0699
10^{-6}	1.0164	1282608	10.5361	10.5672
10^{-2}	1.1547	1283488	10.3520	9.8894
10^{-1}	1.1091	1283808	10.8440	10.5969

Vertices: 512, Number of Outgoing Transitions: 26, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	1.2726	1278064	10.1987	10.3154
10^{-6}	1.0987	1279464	11.2745	10.6300
10^{-2}	1.0980	1279872	10.3819	10.2334
10^{-1}	1.4335	1280016	12.2010	11.1082

Vertices: 512, Number of Outgoing Transitions: 26, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	2.8317	1276440	10.6899	10.8414
10^{-6}	1.3177	1277000	10.8038	10.9364
10^{-2}	1.1545	1277152	11.4884	11.5488
10^{-1}	1.1214	1277208	10.3431	10.4271

Vertices: 512, Number of Outgoing Transitions: 52, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	2.6221	2464664	16.7408	17.3628
10^{-6}	2.3983	2470648	17.8577	17.3655
10^{-2}	2.9024	2472648	17.4577	17.7159
10^{-1}	2.6700	2473400	16.9424	16.9729

Vertices: 512, Number of Outgoing Transitions: 52, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	2.2986	2467776	17.2783	17.5776
10^{-6}	2.3315	2470832	17.1689	17.4705
10^{-2}	2.2997	2471864	18.5313	17.8338
10^{-1}	2.7761	2472256	18.1613	17.3224

Vertices: 512, Number of Outgoing Transitions: 52, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	2.4846	2462736	17.6466	16.3020
10^{-6}	2.2292	2464344	16.8505	17.0691
10^{-2}	2.3480	2464856	16.7904	16.6864
10^{-1}	2.3356	2465048	16.8831	16.7235

Vertices: 512, Number of Outgoing Transitions: 52, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	4.1290	2465896	16.7406	16.9010
10^{-6}	2.4346	2466248	16.8491	17.0449
10^{-2}	2.3025	2466392	17.6820	16.4167
10^{-1}	2.3626	2466448	17.0140	17.2223

Vertices: 512, Number of Outgoing Transitions: 103, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	5.0801	4801816	30.5797	30.3765
10^{-6}	5.4131	4807936	31.0918	32.5726
10^{-2}	5.7495	4809872	30.8827	30.6903
10^{-1}	6.0738	4810592	30.8414	30.3702

Vertices: 512, Number of Outgoing Transitions: 103, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	6.4589	4798368	29.4813	30.2259
10^{-6}	5.8920	4801304	31.1732	30.1112
10^{-2}	6.1924	4802240	30.5585	31.2341
10^{-1}	5.7618	4802584	30.6289	29.8360

Vertices: 512, Number of Outgoing Transitions: 103, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	6.2586	4801872	30.2002	29.7021
10^{-6}	5.7467	4803448	30.5894	31.0917
10^{-2}	5.8835	4803920	31.5345	31.0223
10^{-1}	5.8550	4804088	30.4884	30.8730

Vertices: 512, Number of Outgoing Transitions: 103, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	7.4092	4802344	30.3643	30.2686
10^{-6}	5.9238	4802672	30.7029	32.0089
10^{-2}	6.2521	4802808	30.2577	30.5916
10^{-1}	6.0144	4802864	31.5033	30.4375

Vertices: 512, Number of Outgoing Transitions: 256, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	13.1918	11940672	N/A	N/A
10^{-6}	15.9898	11946824	N/A	N/A
10^{-2}	15.2971	11948752	N/A	N/A
10^{-1}	15.3352	11949456	N/A	N/A

Vertices: 512, Number of Outgoing Transitions: 256, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	16.3449	11945856	N/A	N/A
10^{-6}	15.3214	11948928	N/A	N/A
10^{-2}	16.7030	11949864	N/A	N/A
10^{-1}	16.4426	11950208	N/A	N/A

Vertices: 512, Number of Outgoing Transitions: 256, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	16.7561	11947672	N/A	N/A
10^{-6}	15.9534	11949200	N/A	N/A
10^{-2}	17.5112	11949744	N/A	N/A
10^{-1}	15.7722	11949944	N/A	N/A

Vertices: 512, Number of Outgoing Transitions: 256, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	18.3105	11944872	N/A	N/A
10^{-6}	16.6333	11945192	N/A	N/A
10^{-2}	17.6552	11945280	N/A	N/A
10^{-1}	16.2489	11945312	N/A	N/A

Vertices: 512, Number of Outgoing Transitions: 512, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	29.2841	23908144	N/A	N/A
10^{-6}	32.4629	23913984	N/A	N/A
10^{-2}	32.1413	23916056	N/A	N/A
10^{-1}	30.9576	23916832	N/A	N/A

Vertices: 512, Number of Outgoing Transitions: 512, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	33.9604	23896808	N/A	N/A
10^{-6}	30.9158	23899720	N/A	N/A
10^{-2}	34.0443	23900744	N/A	N/A
10^{-1}	32.6669	23901136	N/A	N/A

Vertices: 512, Number of Outgoing Transitions: 512, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	34.6849	23889360	N/A	N/A
10^{-6}	31.1221	23890848	N/A	N/A
10^{-2}	34.1156	23891344	N/A	N/A
10^{-1}	33.2366	23891528	N/A	N/A

Vertices: 512, Number of Outgoing Transitions: 512, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	36.4159	23902688	N/A	N/A
10^{-6}	31.3989	23903064	N/A	N/A
10^{-2}	34.5132	23903208	N/A	N/A
10^{-1}	33.1103	23903256	N/A	N/A

Vertices: 1023

Vertices: 1023, Number of Outgoing Transitions: 52, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	7.8427	4991448	58.6168	57.6232
10^{-6}	8.9337	4997816	58.4119	57.3419
10^{-2}	9.2529	5000472	55.0021	57.1783
10^{-1}	9.0073	5001008	56.0829	58.0898

Vertices: 1023, Number of Outgoing Transitions: 52, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	9.0655	4994104	55.0649	55.7066
10^{-6}	9.2709	4996928	55.5612	56.2881
10^{-2}	9.9694	4998104	55.3973	55.4900
10^{-1}	10.6333	4998336	56.0184	55.4651

Vertices: 1023, Number of Outgoing Transitions: 52, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	10.6043	4991000	54.7075	54.2913
10^{-6}	8.7229	4992704	55.2278	54.5833
10^{-2}	9.9905	4993416	54.7420	55.7286
10^{-1}	9.5191	4993560	55.9196	55.9996

Vertices: 1023, Number of Outgoing Transitions: 52, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	20.1120	4986792	59.4760	55.0898
10^{-6}	9.2648	4987176	59.0976	58.2981
10^{-2}	10.5559	4987336	56.3348	58.2203
10^{-1}	10.1293	4987368	58.5169	58.6428

Vertices: 1023, Number of Outgoing Transitions: 103, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	17.7542	9722008	N/A	N/A
10^{-6}	19.6730	9728048	N/A	N/A
10^{-2}	20.7443	9730560	N/A	N/A
10^{-1}	21.1419	9731064	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 103, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	20.5142	9721528	N/A	N/A
10^{-6}	20.2830	9724616	N/A	N/A
10^{-2}	22.4271	9725896	N/A	N/A
10^{-1}	20.0986	9726152	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 103, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	22.0643	9711192	N/A	N/A
10^{-6}	20.9082	9712624	N/A	N/A
10^{-2}	22.7772	9713216	N/A	N/A
10^{-1}	20.8840	9713336	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 103, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	32.0730	9721544	N/A	N/A
10^{-6}	20.3567	9721912	N/A	N/A
10^{-2}	22.5374	9722072	N/A	N/A
10^{-1}	20.5520	9722104	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 205, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	37.5035	19367408	N/A	N/A
10^{-6}	40.5454	19373496	N/A	N/A
10^{-2}	40.6982	19376032	N/A	N/A
10^{-1}	39.9403	19376536	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 205, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	42.4705	19355104	N/A	N/A
10^{-6}	40.1504	19358208	N/A	N/A
10^{-2}	42.8471	19359504	N/A	N/A
10^{-1}	41.8003	19359768	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 205, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	43.8436	19366712	N/A	N/A
10^{-6}	41.2950	19368248	N/A	N/A
10^{-2}	43.8403	19368888	N/A	N/A
10^{-1}	41.5880	19369016	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 205, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	55.3887	19364552	N/A	N/A
10^{-6}	39.7800	19364840	N/A	N/A
10^{-2}	44.8401	19364960	N/A	N/A
10^{-1}	40.1583	19364984	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 512, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	82.5123	48363264	N/A	N/A
10^{-6}	98.5603	48369552	N/A	N/A
10^{-2}	99.9398	48372168	N/A	N/A
10^{-1}	99.9411	48372696	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 512, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	108.1544	48374208	N/A	N/A
10^{-6}	100.1259	48377384	N/A	N/A
10^{-2}	100.5867	48378712	N/A	N/A
10^{-1}	101.1974	48378976	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 512, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	106.5735	48370624	N/A	N/A
10^{-6}	100.2492	48372328	N/A	N/A
10^{-2}	108.2881	48373040	N/A	N/A
10^{-1}	102.8106	48373176	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 512, Number of Priorities: 32

ε	TT	Size	VIT	PIT
$None$	119.8714	48361216	N/A	N/A
10^{-6}	100.1333	48361744	N/A	N/A
10^{-2}	103.6187	48361960	N/A	N/A
10^{-1}	99.4520	48362008	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 1023, Number of Priorities: 2

ε	TT	Size	VIT	PIT
$None$	177.2076	96719232	N/A	N/A
10^{-6}	196.1495	96725608	N/A	N/A
10^{-2}	207.2338	96728264	N/A	N/A
10^{-1}	196.2539	96728792	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 1023, Number of Priorities: 4

ε	TT	Size	VIT	PIT
$None$	216.8616	96675392	N/A	N/A
10^{-6}	198.1531	96678280	N/A	N/A
10^{-2}	197.6017	96679488	N/A	N/A
10^{-1}	199.2781	96679728	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 1023, Number of Priorities: 8

ε	TT	Size	VIT	PIT
$None$	221.3773	96705328	N/A	N/A
10^{-6}	204.2478	96706888	N/A	N/A
10^{-2}	208.2147	96707536	N/A	N/A
10^{-1}	204.0472	96707664	N/A	N/A

Vertices: 1023, Number of Outgoing Transitions: 1023, Number of Priorities: 32

ε	**TT**	**Size**	**VIT**	**PIT**
$None$	342.2393	96712264	N/A	N/A
10^{-6}	282.0843	96712672	N/A	N/A
10^{-2}	290.5614	96712840	N/A	N/A
10^{-1}	290.0384	96712880	N/A	N/A