Reachability Switching Games

John Fearnley
University of Liverpool, UK
john.fearnley@liverpool.ac.uk

Martin Gairing
University of Liverpool, UK
gairing@liverpool.ac.uk

Matthias Mnich[1]
Universität Bonn, Germany
and Maastricht University, The Netherlands
mmnich@uni-bonn.de

Rahul Savani
University of Liverpool, UK
rahul.savani@liverpool.ac.uk

Abstract

In this paper, we study the problem of deciding the winner of reachability switching games. We study zero-, one-, and two-player variants of these games. We show that the zero-player case is NL-hard, the one-player case is NP-complete, and that the two-player case is PSPACE-hard and in EXPTIME. For the zero-player case, we also show P-hardness for a succinctly-represented model that maintains the upper bound of NP ∩ coNP. For the one- and two-player cases, our results hold in both the natural, explicit model and succinctly-represented model. We also study the structure of winning strategies in these games, and in particular we show that exponential memory is required in both the one- and two-player settings.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases Deterministic Random Walks, Model Checking, Reachability, Simple Stochastic Game, Switching Systems

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.124

Related Version A full version of this paper with all proofs is available at [8], `https://arxiv.org/abs/1709.08991`.

1 Introduction

A *switching system* (also known as a Propp machine) attempts to replicate the properties of a random system in a deterministic way [14]. It does so by replacing the nodes of a Markov chain with *switching nodes*. Each switching node maintains a queue over its outgoing edges. When the system arrives at the node, it is sent along the first edge in this queue, and that edge is then sent to the back of the queue. In this way, the switching node ensures that, after a large number of visits, each outgoing edge is used a roughly equal number of times.

The Propp machine literature has focussed on *many-token* switching systems and has addressed questions such as how well these systems emulate Markov chains. Recently, Dohrau

[1] Supported by ERC Starting Grant 306465 (BeyondWorstCase) and DFG grant MN 59/4-1.

et. al. [7] initiated the study of *single-token* switching systems and found that the reachability problem raised interesting complexity-theoretic questions. Inspired by that work, we study the question *how hard is it to model check single-token switching systems?* A switching node is a simple example of a fair scheduler, and thus it is natural to consider model checking of switching systems. We already have a good knowledge about the complexity of model checking Markovian systems, but how does this change when we instead use switching nodes?

Our contribution. In this paper, we initiate the study of model checking in switching systems. We focus on *reachability* problems, one of the simplest model checking tasks. This corresponds to determining the winner of a *two-player reachability switching game*. We study zero-, one-, and two-player variants of these games, which correspond to switching versions of Markov chains, Markov decision processes [20], and simple stochastic games [2], respectively.

The main message of the paper is that deciding reachability in one- and two-player switching games is harder than deciding reachability in Markovian systems. Specifically, we show that deciding the winner of a one-player game is NP-complete, and that the problem of deciding the winner of a two-player game is PSPACE-hard and in EXPTIME.

We also study the complexity of zero-player games, where we show hardness results that complement the upper bounds shown in previous work [7]. For the standard model of switching systems, which we call *explicit games*, we are able to show a lower bound of NL-hardness, which is still quite far from the known upper bound of $\mathtt{UP} \cap \mathtt{coUP}$. We also show that if one extends the model by allowing the switching order to be represented in a concise way, then a stronger lower bound of P-hardness can be shown, while still maintaining an $\mathtt{NP} \cap \mathtt{coNP}$ upper bound. We call these concisely represented games *succinct games*, and we also observe that all of our other results, both upper and lower bounds, still apply to succinct games. Our results are summarised in the following table.

	Markovian	Switching (explicit)	Switching (succinct)
0-player	PL-complete[2]	NL-hard; in CLS, in $\mathtt{UP} \cap \mathtt{coUP}$	P-hard; in $\mathtt{NP} \cap \mathtt{coNP}$
1-player	P-complete	NP-complete	NP-complete
2-player	$\mathtt{NP} \cap \mathtt{coNP}$	PSPACE-hard; in EXPTIME	PSPACE-hard; in EXPTIME

For the explicit zero-player case the first bound was an $\mathtt{NP} \cap \mathtt{coNP}$ upper bound given by Dohrau et al. [7], and a PLS upper bound was then given by Karthik [15]. The CLS and $\mathtt{UP} \cap \mathtt{coUP}$ upper bounds, which subsume the two earlier bounds, were given by Gärtner et al. [10], who also produced a $O(1.4143^n)$ algorithm for solving explicit zero-player games. All the other upper and lower bounds in the table are proved in this paper.

Finally, we address the memory requirements of winning strategies in reachability switching games. It is easy to see that winning strategies exist that use exponential memory. These strategies simply remember the current switch configuration of the switching nodes, and their existence can be proved by blowing up a switching game into an exponentially sized reachability game, and then following the positional winning strategies from that reachability game. This raises the question of whether there are winning strategies that use less than

[2] PL, or probabilistic L, is the class of languages recognizable by a polynomial time logarithmic space randomized machine with probability $> 1/2$.

exponential memory. We answer this negatively, by showing that the reachability player may need $\Omega(2^{n/2})$ memory states to win a one-player reachability switching game, and that both players may need to use $\Omega(2^n)$ memory states to win a two-player game.

Related work. Switching games are part of a research thread at the intersection of computer science and physics. This thread has studied zero-player switching systems, also known as *deterministic random walks*, *rotor-router walks*, the *Eulerian walkers model* [19] and *Propp machines* [3–6, 13, 14]. Propp machines have been studied in the context of derandomizing algorithms and pseudorandom simulation, and in particular have received a lot of attention in the context of load balancing [1, 9]. However, most work on Propp machines has focused on how well multi-token switching systems simulate Markov chains. The idea of studying *single-token* reachability should be credited to Dohrau at al. [7].

Katz et al. [16], Groote and Ploeger [12], and others [12, 18, 21], considered *switching graphs*; these are graphs in which certain vertices (switches) have exactly one of their two outgoing edges activated. However, the activation of the alternate edge does not occur when a vertex is traversed by a run; this is the key difference to switching games in this paper.

Markov decision processes [20] and simple stochastic games [2] are important objects of study in *probabilistic model checking*, which is an central topic in the field of formal verification. Probabilistic model checking is now a mature topic, with tools like PRISM [17] providing an accessible interface to the research that has taken place.

2 Preliminaries

A reachability switching game (RSG) is defined by a tuple $(V, E, V_\mathrm{R}, V_\mathrm{S}, V_\mathrm{Swi}, \mathrm{Ord}, s, t)$, where (V, E) is a finite directed graph, and V_R, V_S, V_Swi partition V into *reachability vertices*, *safety vertices*, and *switching vertices*, respectively. The reachability vertices V_R are controlled by the *reachability player*, the safety vertices V_S are controlled by the *safety player*, and the switching vertices V_Swi are not controlled by either player, but instead follow a predefined "switching order". The function Ord defines this *switching order*: for each switching vertex $v \in V_\mathrm{Swi}$, we have that $\mathrm{Ord}(v) = \langle u_1, u_2, \dots, u_k \rangle$ where we have that $(v, u_i) \in E$ for all u_i in the sequence. Note that a particular vertex u may appear more than once in the sequence. The vertices $s, t \in V$ specify *source* and *target* vertices for the game.

A *state* of the game is defined by a tuple (v, C), where v is a vertex in V, and $C : V_\mathrm{Swi} \to \mathbb{N}$ is a function that assigns a number to each switching vertex, which represents how far that vertex has progressed through its switching order. Hence, it is required that $C(u) \leq |\mathrm{Ord}(v)| - 1$, since the counts specify an index to the sequence $\mathrm{Ord}(v)$.

When the game is at a state (v, C) with $v \in V_\mathrm{R}$ or $v \in V_\mathrm{S}$, then the respective player chooses an outgoing edge at v, and the count function does not change. For states (v, C) with $v \in V_\mathrm{Swi}$, the successor state is determined by the count function. More specifically, we define $\mathrm{Upd}(v, C) : V_\mathrm{Swi} \to \mathbb{N}$ so that for each $u \in V_\mathrm{Swi}$ we have $\mathrm{Upd}(v, C)(u) = C(u)$ if $v \neq u$, and $\mathrm{Upd}(v, C)(u) = (C(u) + 1) \bmod |\mathrm{Ord}(u)|$ otherwise. This function increases the count at v by 1, and wraps around to 0 if the number is larger than the length of the switching order at v. Then, the successor state of (v, C), denoted as $\mathrm{Succ}(v, C)$ is $(u, \mathrm{Upd}(v, C))$, where u is the element at position $C(v)$ in $\mathrm{Ord}(v)$.

A *play* of the game is a (potentially infinite) sequence of states $(v_1, C_1), (v_2, C_2), \dots$ with the following properties:

1. $v_1 = s$ and $C_1(v) = 0$ for all $v \in V_\mathrm{Swi}$;
2. If $v_i \in V_\mathrm{R}$ or $v_i \in V_\mathrm{S}$ then $(v_i, v_{i+1}) \in E$ and $C_i = C_{i+1}$;

3. If $v_i \in V_{\text{Swi}}$ then $(v_{i+1}, C_{i+1}) = \text{Succ}(v_i, C_i)$;
4. If the play is finite, then the final state (v_n, C_n) must either satisfy $v_n = t$, or v_n must have no outgoing edges.

A play is *winning for the reachability player* if it is finite and the final state is the target vertex. A (deterministic, history dependent) *strategy for the reachability player* is a function that maps each play prefix $(v_1, C_1), (v_2, C_2), \dots, (v_k, C_1)$, with $v_k \in V_{\text{R}}$, to an outgoing edge of v_k. A play $(v_1, C_1), (v_2, C_2), \dots$ is *consistent* with a strategy if, whenever $v_i \in V_{\text{R}}$, we have that (v_i, v_{i+1}) is the edge chosen by the strategy. Strategies for the safety player are defined analogously. A strategy is *winning* if all plays consistent with it are winning.

The representation of the switching order. Recall that $\text{Ord}(v) = \langle u_1, u_2, \dots, u_k \rangle$ gives a sequence of outgoing edges for every switching vertex. We consider two possible ways of representing $\text{Ord}(v)$ in this paper. In *explicit* RSGs, $\text{Ord}(v)$ is represented by simply writing down the sequence $\langle u_1, u_2, \dots, u_k \rangle$.

We also consider games in which $\text{Ord}(v)$ is written down in a more concise way, which we call *succinct* RSGs. In these games, for each switching vertex v, we have a sequence of pairs $\langle (u_1, t_1), (u_2, t_2), \dots, (u_k, t_k) \rangle$, where each u_i is a vertex with $(v, u_i) \in E$, and each t_i is a natural number. The idea is that $\text{Ord}(v)$ should contain t_1 copies of u_1, followed by t_2 copies of u_2, and so on. So, if $\text{Rep}(u, t)$ gives the sequence containing t copies of u, and if $\cdot$ represents sequence concatenation, then $\text{Ord}(v) = \text{Rep}(u_1, t_1) \cdot \text{Rep}(u_2, t_2) \cdot \dots \cdot \text{Rep}(u_k, t_k)$. Any explicit game can be written down in the succinct encoding by setting all $t_i = 1$. Note, however, that in a succinct game $\text{Ord}(v)$ may have exponentially many elements, even if the input size is polynomial, since each t_i is represented in binary.

3 One-player reachability switching games

In this section we consider one-player RSGs, i.e., where $V_{\text{S}} = \emptyset$.

3.1 Containment in NP

We show that deciding whether the reachability player wins a one-player RSG is in NP. Our proof holds for both explicit and succinct games. The proof uses *controlled switching flows*. These extend the idea of switching flows, which were used by Dohrau et al. [7] to show containment of the zero-player reachability problem in $\texttt{NP} \cap \texttt{coNP}$.

Controlled switching flow. A *flow* is a function $F : E \to \mathbb{N}$ that assigns a natural number to each edge in the game. For each vertex v, we define $\text{Bal}(F, v) = \sum_{(v,u) \in E} F(v, u) - \sum_{(w,v) \in E} F(w, v)$, which is the difference between the outgoing and incoming flow at v. For each switching node $v \in V_{\text{Swi}}$, let $\text{Succ}(v)$ denote the set of vertices that appear in $\text{Ord}(v)$, and for each index $i \leq |\text{Ord}(v)|$ and each vertex $u \in \text{Succ}(v)$, let $\text{Out}(v, i, u)$ be the number of times that u appears in the first i entries of $\text{Ord}(v)$. In other words, $\text{Out}(v, i, u)$ gives the amount of flow that should be sent to u if we send exactly i units of flow into v.

A flow F is a *controlled switching flow* if it satisfies the following constraints:

- The source vertex s satisfies $\text{Bal}(F, s) = 1$, and the target vertex t satisfies $\text{Bal}(F, t) = -1$.
- Every vertex v other than s or t satisfies $\text{Bal}(F, v) = 0$.
- Let $v \in V_{\text{Swi}}$ be a switching node, $k = |\text{Ord}(v)|$, and let $I = \sum_{(u,v) \in E} F(u, v)$ be the total amount of flow incoming to v. Define p to be the largest integer such that $p \cdot k \leq I$ (which may be 0), and $q = I \bmod k$. For every vertex $w \in \text{Succ}(v)$ we have that $F(v, w) = p \cdot \text{Out}(v, k, w) + \text{Out}(v, q, w)$.

The first two constraints ensure that F is a flow from s to t, while the final constraint ensures that the flow respects the switching order at each switching node. Note that there are no constraints on how the flow is split at the nodes in V_{R}. For each flow F, we define the size of F to be $\sum_{e \in E} F(e)$. A flow of size k can be written down using at most $|E| \cdot \log k$ bits.

Marginal strategies. A *marginal* strategy for the reachability player is defined by a function $M : E \to \mathbb{N}$, which assigns a target number to each outgoing edge of the vertices in V_{R}. The strategy ensures that each edge e is used no more than $M(e)$ times. That is, when the play arrives at a vertex $v \in V_{\mathrm{R}}$, the strategy checks how many times each outgoing edge of v has been used so far, and selects an arbitrary outgoing edge e that has been used strictly less than $M(e)$ times. If there is no such edge, then the strategy is undefined.

Observe that a controlled switching flow defines a marginal strategy for the reachability player. We prove that this strategy always reaches the target.

▶ **Lemma 1.** *If a one-player RSG has a controlled switching flow F, then any corresponding marginal strategy is winning for the reachability player.*

In the other direction, if the reachability player has a winning strategy, then there exists a controlled switching flow, and we can give an upper bound on its size.

▶ **Lemma 2.** *If the reachability player has a winning strategy for a one-player RSG , then that game has a controlled switching flow F, and the size of F is at most $n \cdot l^n$, where n is the number of nodes in the game and $l = \max_{v \in V_{Swi}} |\operatorname{Ord}(v)|$.*

▶ **Corollary 3.** *If the reachability player has a winning strategy for a one-player RSG, then he also has a marginal winning strategy.*

Finally, we can show that solving a one-player RSG is in `NP`.

▶ **Theorem 4.** *Deciding the winner of an explicit or succinct one-player RSG is in* `NP`*.*

3.2 NP-hardness

In this section we show that deciding the winner of a one-player RSG is `NP`-hard. Our construction will produce an explicit RSG, so we obtain `NP`-hardness for both explicit and succinct games. We reduce from 3SAT. Throughout this section, we will refer to a 3SAT instance with variables $x_1, x_2, \dots, x_n$, and clauses $C_1, C_2, \dots, C_m$. It is well-known [22, Thm. 2.1] that 3SAT remains `NP`-hard even if all variables appear in at most three clauses. We make this assumption during our reduction.

Overview. The idea behind the construction is that the player will be asked to assign values to each variable. Each variable x_i has a corresponding vertex that will be visited 3 times during the game. Each time this vertex is visited, the player will be asked to assign a value to x_i in a particular clause C_j. If the player chooses an assignment that *does not* satisfy C_j, then the game records this by incrementing a counter. If the counter corresponding to any clause C_j is incremented to three (or two if the clause only has two variables), then the reachability player immediately loses, since the chosen assignment fails to satisfy C_j.

The problem with the idea presented so far is that there is no mechanism to ensure that the reachability player chooses a consistent assignment to the same variable. Since each variable x_i is visited three times, there is nothing to stop the reachability player from choosing contradictory assignments to x_i on each visit. To address this, the game also counts

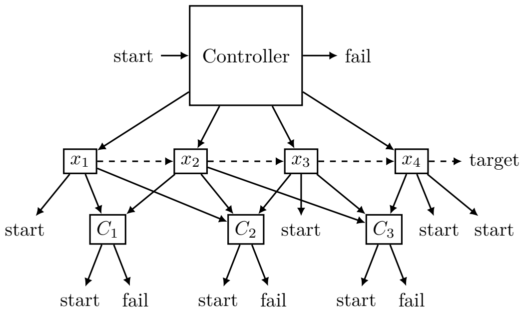

Figure 1 Overview of our construction for one player for the example formula $C_1 \wedge C_2 \wedge C_3 = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4)$. Note that the negations of variables in the formula are not relevant for this high-level view; they will feature in the clause gadgets as explained below. The edges for the variable phase are solid, and the edges for the verification phase are dashed.

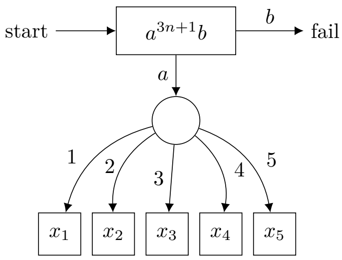

Figure 2 The control gadget.

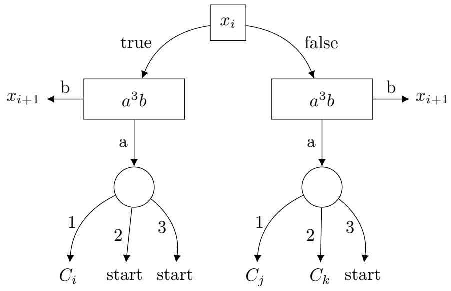

Figure 3 A variable gadget.

how many times each assignment is chosen for x_i. At the end of the game, if the reachability player has not already lost by failing to satisfy the formula, the game is configured so that the target is only reachable if the reachability player chose a consistent assignment. A high-level overview of the construction for an example formula is given in Fig. 1.

The control gadget. The sequencing in the construction is determined by the control gadget, which is shown in Fig. 2. In our diagramming notation, square vertices belong to the reachability player. Circle vertices are switching nodes, and the switching order of each switching vertex is labelled on its outgoing edges. Our diagrams also include *counting gadgets*, which are represented as non-square rectangles that have labelled output edges. The counting gadget is labelled by a sequence over these outputs, with the idea being that if the play repeatedly reaches the gadget, then the corresponding output sequence will be produced. In Fig. 2 the gadget is labelled by $a^{3n+1}b$, which means the first $3n+1$ times the gadget is used the token will be moved along the a edge, and the $3n+2$nd time the gadget is used the token will be moved along the b edge. This gadget can be easily implemented by a switching node that has $3n+2$ outgoing edges, the first $3n+1$ of which go to a, while the $3n+2$nd edge goes to b. We use gadgets in place of this because it simplifies our diagrams.

The control gadget has two phases. In the *variable phase*, each variable gadget, represented by the vertices x_1 through x_n is used exactly 3 times, and thus overall the gadget will be used $3n$ times. This is accomplished by a switching node that ensures that each variable is used 3 times. After each variable gadget has been visited 3 times, the control gadget then sends the token to the x_1 variable gadget for the *verification phase* of the game. In this phase, the reachability player must prove that he gave consistent assignments to all variables.

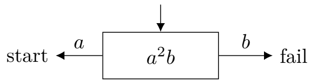

Figure 4 A gadget for a clause with three variables.

If the control gadget is visited $3n + 2$ times, then the token will be moved to the *fail* vertex. This vertex has no outgoing edges, and thus is losing for the reachability player.

The variable gadgets. Each variable x_i is represented by a variable gadget, which is shown in Fig. 3. This gadget will be visited 3 times in total during the variable phase, and each time the reachability player must choose either the true or false edges at the vertex x_i. In either case, the token will then pass through a counting gadget, and then move to a switching vertex which either moves the token to a clause gadget, or back to the start vertex.

It can be seen that the gadget is divided into two almost identical branches. One corresponds to a true assignment to x_i, and the other to a false assignment to x_i. The clause gadgets are divided between the two branches of the gadget. In particular, a clause appears on a branch if and only if the variable appears in that clause and the choice made by the reachability player *fails* to satisfy the clause. So, the clauses in which x_i appears positively appear on the false branch of the gadget, while the clauses in which x_i appears negatively appear on the true branch.

The switching vertices each have exactly three outgoing edges. These edges use an arbitrary order over the clauses assigned to the branch. If there are fewer than 3 clauses on a particular branch, the remaining edges of the switching node go back to the start vertex. Note that this means that a variable can be involved with fewer than three clauses.

The counting gadgets will be used during the verification phase of the game, in which the variable player must prove that he has chosen consistent assignments to each of the variables. Once each variable gadget has been used 3 times, the token will be moved to x_1 by the control gadget. If the reachability player has used the same branch three times, then he can choose that branch, and move to x_2, which again has the same property. So, if the reachability player gives a consistent assignment to all variables, he can eventually move to x_n, and then on to x_{n+1}, which is the target vertex of the game. Since, as we will show, there is no other way of reaching x_{n+1}, this ensures that the reachability player must give consistent assignments to the variables in order to win the game.

The clause gadgets. Each clause C_j is represented by a clause gadget, an example of which is shown in Fig. 4. The gadget counts how many variables have failed to satisfy the corresponding clause. If the number of times the gadget is visited is equal to the number of variables involved with the clause, then the game moves to the fail vertex, and the reachability player immediately loses. In all other cases, the token moves back to the start vertex.

Correctness. The following lemma shows that the reachability player wins the one-player RSG if and only if the 3SAT instance is satisfiable.

▶ **Lemma 5.** *The reachability player wins the one-player RSG if and only if the 3SAT instance is satisfiable.*

Note that our game can be written down as an explicit game, so our lower bound applies to both explicit and succinct games. Hence, we have the following theorem.

▶ **Theorem 6.** *Deciding the winner of an explicit or succinct one-player RSG is* `NP`*-hard.*

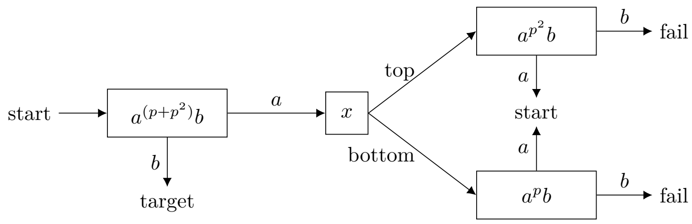

Figure 5 One-player memory lower bound construction.

3.3 Memory requirements of winning strategies in one-player games

Consider the game shown in Fig. 5, which takes as input a parameter p that we will fix later. The only control state for the player is x. By construction, x will be visited $p + p^2$ times. Each time, the player must choose either the top or bottom edge. If the player uses the top edge strictly more than p^2 times, or the bottom edge strictly more than p times, then he will immediately lose the game. If the player does not lose the game in this way, then after $p^2 + p$ rounds the target will be reached, and the player will win the game.

The player has an obvious winning strategy: use the top edge p^2 times and the bottom edge p times. Intuitively, there are two ways that the player could implement the strategy. (1) Use the bottom edge p times, and then use the top edge p^2 times. This approach uses p memory states to count the number of times the bottom edge has been used. (2) Use the bottom edge once, use the top edge p times, and then repeat. This approach uses p memory states to count the number of times the top state has been used after each use of the bottom edge. We can prove that one cannot do significantly better.

▶ **Lemma 7.** *The reachability player must use at least $p - 1$ memory states to win the game shown in Fig. 5.*

Setting $p = 2^{n/2}$ gives us our lower bound. Even though p is exponential, it is possible to create an explicit switching gadget that produces the sequence $a^{2^n}b$ with n switching nodes.

▶ **Lemma 8.** *For all $x \in \mathbb{N}$ there is an explicit switching gadget of size $\log_2(x)$ with output $a^x b$.*

▶ **Theorem 9.** *The number of memory states needed in an explicit one-player RSG is $\Omega(2^{\frac{n}{2}})$.*

4 Two-player reachability switching games

4.1 Containment in EXPTIME

We first observe that solving a two-player RSG lies in `EXPTIME`. This can be proved easily, either by blowing the game up into an exponentially sized reachability game, or equivalently, by simulating the game on an alternating polynomial-space Turing machine.

▶ **Theorem 10.** *Deciding the winner of an RSG is in* `EXPTIME`.

4.2 PSPACE-hardness

We show that deciding the winner of an explicit two-player RSG is `PSPACE`-hard, by reducing *true quantified boolean formula* (TQBF), the canonical `PSPACE`-complete problem, to our problem. Throughout this section we will refer to a TQBF instance $\exists x_1 \forall x_2 \ldots \exists x_{n-1} \forall x_n \cdot \phi(x_1, x_2, \ldots, x_n)$, where ϕ denotes a boolean formula given in negation normal form, which requires that negations are only applied to variables, and not sub-formulas. The problem is to decide whether this formula is true.

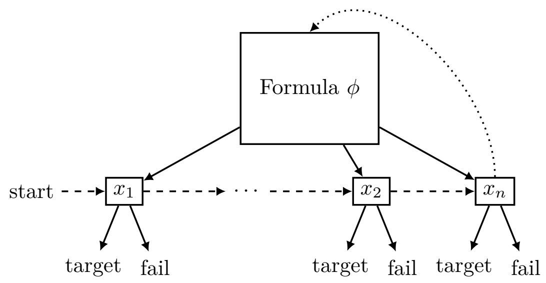

Figure 6 High-level overview of our construction for two players. The dashed lines between variables are part of the first, quantifier phase; the dotted line from variable x_n to the Formula is the transition between phases, and the solid edges are part of the second, formula phase.

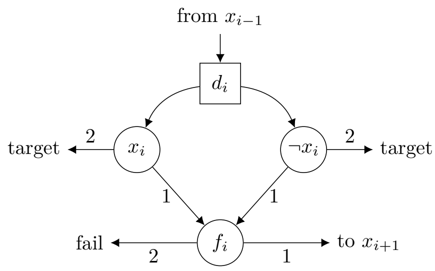

Figure 7 The initialization gadget for an existentially quantified variable x_i.

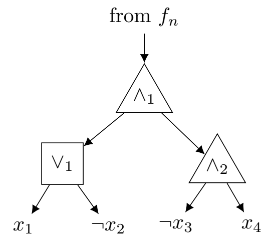

Figure 8 The formula phase game for the formula $(x_1 \vee \neg x_2) \wedge \neg x_3 \wedge x_4$.

Overview. We will implement the TQBF formula as a game between the reachability player and the safety player. This game will have two phases. In the *quantifier phase*, the two players assign values to their variables in the order specified by the quantifiers. In the *formula phase*, the two players determine whether ϕ is satisfied by these assignments by playing the standard model-checking game for propositional logic. The target state of the game is reached if and only if the model checking game determines that the formula is satisfied. This high-level view of our construction is depicted in Fig. 6.

The quantifier phase. Each variable in the TQBF formula will be represented by an *initialization gadget*. The initialization gadget for an existentially quantified variable is shown in Fig. 7. The gadget for a universally quantified variable is almost identical, but the state d_i is instead controlled by the safety player.

During the quantifier phase, the game will start at d_1, and then pass through the gadgets for each of the variables in sequence. In each gadget, the controller of d_i must move to either x_i or $\neg x_i$. In either case, the corresponding switching node moves the token to f_i, which then subsequently moves the token on to the gadget for x_{i+1}.

The important property to note here is that once the player has made a choice, any subsequent visit to x_i or $\neg x_i$ will end the game. Suppose that the controller of d_i chooses to move to x_i. If the token ever arrives at x_i a second time, then the switching node will move to the target vertex and the reachability player will immediately win the game. If the token ever arrives at $\neg x_i$ the token will move to f_i and then on to the fail vertex, and the Safety player will immediately win the game. The same property holds symmetrically if the controller of d_i chooses $\neg x_i$ instead. In this way, the controller of d_i selects an assignment to x_i. Hence, the reachability player assigns values to the existentially quantified variables, and the safety player assigns values to the universally quantified variables.

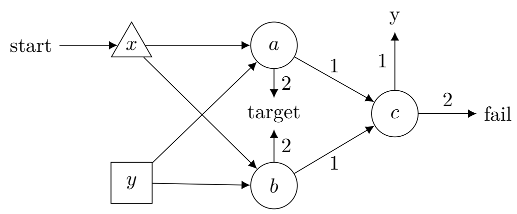

Figure 9 An RSG in which the reachability player needs to use memory.

The formula phase. Once the quantifier phase has ended, the game moves into the formula phase. In this phase the two players play a game to determine whether ϕ is satisfied by the assignments to the variables. This is the standard model checking game for first order logic. The players play a game on the parse tree of the formula, starting from the root. The reachability player controls the $\vee$ nodes, while the safety player controls the $\wedge$ nodes (recall that the game is in negation normal form, so there are no internal $\neg$ nodes.) Each leaf is either a variable or its negation, which in our game are represented by the x_i and $\neg x_i$ nodes in the initialization gadgets. An example of this game is shown in Fig. 8. In our diagramming notation, nodes controlled by the safety player are represented by triangles.

Intuitively, if ϕ is satisfied by the assignment to $x_1, \dots, x_n$, then no matter what the safety player does, the reachability player is able to reach a leaf node corresponding to a true assignment, and as mentioned earlier, he will then immediately win the game. Conversely, if ϕ is not satisfied, then no matter what the reachability player does, the safety player can reach a leaf corresponding to a false assignment, and then immediately win the game.

▶ **Lemma 11.** *The reachability player wins if and only if the QBF formula is true.*

Since we have shown the lower bound for explicit games, we also get the same lower bound for succinct games as well. We have shown the following theorem.

▶ **Theorem 12.** *Deciding the winner of an explicit or succinct RSG is* `PSPACE`*-hard.*

Note that all runs of the game have polynomial length, a property that is not shared by all RSGs. This gives us the following corollary.

▶ **Corollary 13.** *Deciding the winner of a polynomial-length RSG is* `PSPACE`*-complete.*

4.3 Memory requirements for two player games

We can show even stronger memory lower bounds in two-player games compared to one-player games. Fig. 9 shows a simple gadget that forces the reachability player to use memory. The game starts by allowing the safety player to move the token from x to either a or b. Whatever the choice, the token then moves to c and then on to y. At this point, if the reachability player moves the token to the node chosen by the safety player, then the token will arrive at the target node and the reachability player will win. If the reachability player moves to the other node, the token will move to c for a second time, and then on to the fail vertex, which is losing for the reachability player. Thus, every winning strategy of the reachability player must remember the choice made by the safety player.

We can create a similar gadget that forces the safety player to use memory by swapping the players. In the modified gadget, the safety player has to choose the vertex *not chosen* by the reachability player. Thus, in an RSG, winning strategies for both players need to use memory. By using n copies of the memory gadget, we can show the following lower bound.

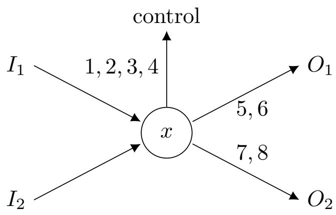

Figure 10 An AND-gate of depth 2.

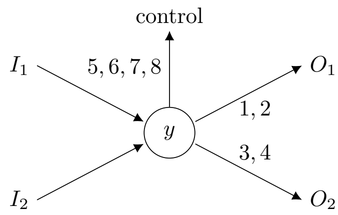

Figure 11 An OR-gate of depth 2.

▶ **Lemma 14.** *In an explicit or succinct RSG, winning strategies for both players may need to use 2^n memory states, where n is the number of switching nodes.*

5 Zero-player reachability switching games

5.1 Explicit zero-player games

We show that deciding the winner of an explicit zero-player game is NL-hard. To do this, we reduce from the problem of deciding s-t connectivity in a directed graph. The idea is to make every node in the graph a switching node. We then begin a walk from s. If, after $|V|$ steps we have not arrived at t, we go back to s and start again. So, if there is a path from s to t, then the switching nodes must eventually send the token along that path.

Formally, given a graph (V, E), we produce a zero-player RSG played on $V \times V \cup \{\mathsf{fin}\}$, where the second component of each state is a counter that counts up to $|V|$. Every vertex is a switching node, the start vertex is $(s, 1)$, and the target vertex is fin. Each vertex (v, k) with $v \neq t$ and $k < |V|$ has outgoing edges to $(u, k + 1)$ for each outgoing edge $(v, u) \in E$. Each vertex $(v, |V|)$ with $v \neq t$ has a single outgoing edge to $(s, 1)$. Every vertex (t, k) with $1 \leq k \leq |V|$ has a single outgoing edge to fin. This game can be constructed in logarithmic space by looping over each element in $V \times V$ and producing the correct outgoing edges.

▶ **Theorem 15.** *Deciding the winner of an explicit zero-player RSG is* NL*-hard under logspace reductions.*

5.2 Succinct games

Deciding reachability for succinct zero-player games still lies in NP ∩ coNP. This can be shown using essentially the same arguments that were used to show NP ∩ coNP containment for explicit games [7]. The fact that the problem lies in NP follows from Theorem 4, since every succinct zero-player game is also a succinct one-player game, and so a switching flow can be used to witness reachability. To put the problem in coNP, one can follow the original proof given by Dohrau et al. [7, Theorem 3] for explicit games. This proof condenses all losing and infinite plays into a single failure state, and then uses a switching flow to witness reachability for that failure state. Their transformation uses only the graph structure of the game, and not the switching order, and so it can equally well be applied to succinct games.

In contrast to explicit games, we can show a stronger lower bound of P-hardness for succinct games. We will reduce from the problem of evaluating a boolean circuit (the *circuit value problem*), which is one of the canonical P-complete problems. We will assume that the circuit has fan-in and fan-out 2, that all gates are either AND-gates or OR-gates, and that the circuit is *synchronous*, meaning that the outputs of the circuit have depth 1, and all gates at depth i get their inputs from gates of depth exactly $i + 1$. This is Problem A.1.6

"Fanin 2, Fanout 2 Synchronous Alternating Monotone CVP" of Greenlaw et al. [11]. We will reduce from the following decision problem: for a given input bit-string $B \in \{0,1\}^n$, and a given output gate g, is g evaluated to true when the circuit is evaluated on B?

Boolean gates. We will simulate the gates of the circuit using switching nodes. A gate at depth $i > 1$ is connected to exactly two gates of depth $i+1$ from which it gets its inputs, and exactly two gates at depth $i-1$ to which it sends its output. If a gate evaluates to true, then it will send a *signal* to the output-gates, by sending the token to that gate's gadget. More precisely, for a gate of depth $i > 1$, the following signals are sent. If the gate evaluates to true, then the gate will send the token exactly 2^{i-1} times to each output gate. If the gate evaluates to false, then the gate will send the token exactly 0 times to each output gate. So the number of signals sent by a gate grows exponentially with the depth of that gate.

Fig. 10 shows our construction for an AND-gate of depth 2. It consists of a single switching node (with a succinct order). I_1 and I_2 are two input edges that come from the two inputs to this gate, and O_1 and O_2 are two output edges that go to the outputs of this gate. The control state is a special state that drives the construction, which will be described later. The switching order was generated by the following rules. For a gate at depth i, the switching order of an AND-gate is defined so that the first 2^i positions in the switching order go to control, the next 2^{i-1} positions in the switching order go to O_1, and the final 2^{i-1} positions in the switching order go to O_2. Observe that this switching order captures the behavior of an AND-gate. If the gadget receives 2^i signals from both inputs, then it sends 2^{i-1} signals to both outputs. On the other hand, if at least one of the two inputs sends no signals, then the gadget sends no signals to the outputs.

The same idea is used to implement OR-gates. Fig. 11 shows the construction for an OR-gate of depth 2. For an OR-gate of depth i we have that the first 2^{i-1} positions in the switching order go to O_1, the next 2^{i-1} positions in the switching order go to O_2, and the final 2^i positions in the switching order go to control. These conditions simulate an OR-gate. If either of the inputs produces 2^i input signals, then 2^{i-1} signals are sent to both outputs. If both inputs produce no signals, then no signals are sent to either output.

The control state and the depth 1 gates. Suppose that the inputs to the circuit are at depth d. The control state is a single switching node that has the following switching order. Each input edge to a gate at depth d refers to some bit contained in the bit-string B. The control state sends 2^d inputs using that edge if that bit is true, and 0 inputs using that edge if that bit is false. Once those signals have been sent, the control state moves the token to an absorbing failure state. The token begins at the control state.

Each gate at depth 1 is represented by a single state, and has the same structure and switch configuration as the gates at depth $i > 1$. The only difference is the destination of the edges O_1 and O_2. The gate g (which we must evaluate) sends all outputs to an absorbing target state. All other gates send all outputs back to the control state.

▶ **Lemma 16.** *The token reaches the target state if and only if the gate g evaluates to true when the circuit is evaluated on the bit-string B.*

Since these gadgets use exponential switching orders, this construction would have exponential size if written down in the explicit format. Note, however, that all of the switching orders can be written down in the succinct format in polynomially many bits. Moreover, the construction has exactly one switching state for each gate in the circuit, and three extra states for the control, target, and failure nodes. Every state in the construction can be created using only the inputs and outputs of the relevant gate in the circuit, which means that the reduction can be carried out in logarithmic space. Thus, we have the following.

▶ **Theorem 17.** *Deciding the winner of a succinct zero-player RSG is* P*-hard under logspace reductions.*

References

1. Hoda Akbari and Petra Berenbrink. Parallel rotor walks on finite graphs and applications in discrete load balancing. In *Proc. of SPAA*, pages 186–195, 2013.
2. Anne Condon. The complexity of stochastic games. *Inf. Comput.*, 96(2):203–224, 1992.
3. Joshua Cooper, Benjamin Doerr, Tobias Friedrich, and Joel Spencer. Deterministic random walks on regular trees. *Random Structures Algorithms*, 37(3):353–366, 2010.
4. Joshua Cooper, Benjamin Doerr, Joel Spencer, and Gábor Tardos. Deterministic random walks on the integers. *European J. Combin.*, 28(8):2072–2090, 2007.
5. Joshua Cooper and Joel Spencer. Simulating a random walk with constant error. *Combin. Probab. Comput.*, 15(6):815–822, 2006.
6. Benjamin Doerr and Tobias Friedrich. Deterministic random walks on the two-dimensional grid. *Combin. Probab. Comput.*, 18(1-2):123–144, 2009.
7. Jérôme Dohrau, Bernd Gärtner, Manuel Kohler, Jiří Matoušek, and Emo Welzl. ARRIVAL: a zero-player graph game in $\mathtt{NP} \cap \mathtt{coNP}$. In *A journey through discrete mathematics*, pages 367–374. Springer, Cham, 2017.
8. John Fearnley, Martin Gairing, Matthias Mnich, and Rahul Savani. Reachability switching games. *CoRR*, abs/1709.08991, 2017. `arXiv:1709.08991`.
9. Tobias Friedrich, Martin Gairing, and Thomas Sauerwald. Quasirandom load balancing. *SIAM J. Comput.*, 41(4):747–771, 2012.
10. Bernd Gärtner, Thomas Dueholm Hansen, Pavel Hubáček, Karel Král, Hagar Mosaad, and Veronika Slívová. ARRIVAL: Next stop in CLS. In *Proc. of ICALP*, 2018. To appear.
11. Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to parallel computation: P-completeness theory.* The Clarendon Press, Oxford University Press, New York, 1995.
12. Jan Friso Groote and Bas Ploeger. Switching graphs. *Internat. J. Found. Comput. Sci.*, 20(5):869–886, 2009.
13. Alexander E. Holroyd, Lionel Levine, Karola Mészáros, Yuval Peres, James Propp, and David B. Wilson. Chip-firing and rotor-routing on directed graphs. In *In and out of equilibrium. 2*, volume 60 of *Progr. Probab.*, pages 331–364. Birkhäuser, Basel, 2008.
14. Alexander E. Holroyd and James Propp. Rotor walks and Markov chains. In *Algorithmic probability and combinatorics*, volume 520 of *Contemp. Math.*, pages 105–126. Amer. Math. Soc., Providence, RI, 2010.
15. Karthik C. S. Did the train reach its destination: The complexity of finding a witness. *Inf. Process. Lett.*, 121:17–21, 2017.
16. Bastian Katz, Ignaz Rutter, and Gerhard Woeginger. An algorithmic study of switch graphs. *Acta Inform.*, 49(5):295–312, 2012.
17. Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of CAV*, pages 585–591, 2011.
18. Christoph Meinel. Switching graphs and their complexity. In *Proc. MFCS 1989*, volume 379 of *Lecture Notes Comput. Sci.*, pages 350–359, 1989.
19. Vyatcheslav B. Priezzhev, Deepak Dhar, Abhishek Dhar, and Supriya Krishnamurthy. Eulerian walkers as a model of self-organized criticality. *Phys. Rev. Lett.*, 77(25):5079, 1996.
20. Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

21 Klaus Reinhardt. The simple reachability problem in switch graphs. In *Proc. of SOFSEM*, pages 461–472, 2009.
22 Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Appl. Math.*, 8(1):85–89, 1984.