Logical Methods in Computer Science
Volume 17, Issue 2, 2021, pp. 10:1–10:29
https://lmcs.episciences.org/

Submitted May 03, 2019
Published Apr. 22, 2021

REACHABILITY SWITCHING GAMES

JOHN FEARNLEY [a], MARTIN GAIRING [a], MATTHIAS MNICH [b], AND RAHUL SAVANI [a]

[a] University of Liverpool, Liverpool, United Kingdom
e-mail address: john.fearnley@liverpool.ac.uk, gairing@liverpool.ac.uk, rahul.savani@liverpool.ac.uk

[b] Hamburg University of Technology, Institute for Algorithms and Complexity, Hamburg, Germany
e-mail address: matthias.mnich@tuhh.de

Abstract. We study the problem of deciding the winner of reachability switching games for zero-, one-, and two-player variants. Switching games provide a deterministic analogue of stochastic 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 $\mathtt{NP} \cap \mathtt{coNP}$. For the one- and two-player cases, our results hold in both the natural, explicit model and succinctly-represented model. Our results show that the switching variant of a game is harder in complexity-theoretic terms than the corresponding stochastic version.

1. Introduction

A *switching system* (also known as a Propp machine) attempts to replicate the properties of a random system in a deterministic way [HP10]. 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 et al. [DGK+17] 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?

Key words and phrases: Deterministic Random Walks, Model Checking, Reachability, Simple Stochastic Game, Switching Systems.

A preliminary version of this paper appeared at ICALP 2018 [FGMS18].

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.23638/LMCS-17(2:10)2021

Our contribution. All work so far has studied zero-player switching systems. In this paper, we initiate the study of model checking in switching systems, which naturally leads to one- and two-player switching systems. We focus on one-token *reachability* problems, one of the simplest model checking tasks for a switching system. 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 [Put94], and simple stochastic games [Con92], respectively. Our results are summarised in Table 1.

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. 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 UP $\cap$ coUP [GHH+18]. 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. We call these concisely represented games *succinct games*, and we are also able to show upper bounds for succinct zero-player games that match the known upper bounds for explicit zero-player games. Furthermore, all of our other results for one and two-player games, both upper and lower bounds, still apply to succinct games.

	Markovian		Switching (explicit)		Switching (succinct)	
0-pl.	PL-compl.[1]		NL-hard	(Thm. 5.1)	P-hard	(Thm. 5.3)
			in UEOPL, CLS, and UP $\cap$ coUP	[GHH+18]	in UEOPL and CLS	(Thm. 5.10)
1-pl.	P-compl.	[PT87]	NP-compl.	(Thm. 3.4 and 3.8)	NP-compl.	(Thm. 3.4 and 3.8)
2-pl.	AUC-SPACE($\log(n)$)-compl.[2]	[Con89]	PSPACE-hard	(Thm. 4.3)	PSPACE-hard	(Thm. 4.3)
	in NP $\cap$ coNP	[Con92]	in EXPTIME	(Thm. 4.1)	in EXPTIME	(Thm. 4.1)

Table 1: A summary of our results.

For the explicit zero-player case the first bound was an NP $\cap$ coNP upper bound given by Dohrau et al. [DGK+17], and a PLS upper bound was then given by Karthik [Kar17]. The UEOPL, CLS, and UP $\cap$ coUP upper bounds, which subsume the two earlier bounds, were given by Gärtner et al. [GHH+18], who also produced a $O(1.4143^n)$-time 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,

[1]PL, or probabilistic L, is the class of languages recognizable by a polynomial-time logarithmic-space randomized machine with probability $> 1/2$; there is a straight-forward polynomial-time inter-reduction between this decision problem for polynomial-time logarithmic-space randomized machines and Markov chains.

[2]AUC-SPACE($\log(n)$) is the class of languages accepted by log-space bounded randomized alternating Turing machines. It was defined by Condon [Con89].

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 exponential memory. We answer this negatively, by showing that the reachability player may need $2^{\Omega(n)}$ memory states to win a one-player reachability switching game, and that both players may need to use $2^{n/6}$ memory states to win a two-player game.

Related work. While we are the first to study switching games with multiple players, zero-player switching systems (so far mainly with multiple tokens) are part of a research thread at the intersection of computer science and physics. These zero-player switching systems are also known as *deterministic random walks*, *rotor-router walks*, the *Eulerian walkers model* [PDDK96] and *Propp machines* [HLM+08, CDST07, CDFS10, DF09, CS06, HP10]. These systems 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 [FGS12, AB13]. Most work on switching systems has focused on how well multi-token switching systems simulate Markov chains.

The idea of studying *single-token* reachability is due to Dohrau et al. [DGK+17], who introduced the problem under the name Arrival. Subsequent work has shown that Arrival lies in the complexity classes `PLS` [Kar17], `CLS` [GHH+18, FGHS21], and `UEOPL` [FGMS20]. Recently, a deterministic sub-exponential algorithm has been given for solving the Arrival problem [GHH21].

We study model checking for single-token switching systems. There is extensive literature on model checking stochastic systems, known as *probabilistic model checking*, which is an central topic in the field of formal verification. Markov decision processes [Put94] and simple stochastic games [Con92] are important objects of study in probabilistic model checking. Probabilistic model checking is now a mature topic, with tools like PRISM [KNP11] providing an accessible interface to the research that has taken place.

Other notions of switching nodes have been studied. Katz et al. [KRW12], Groote and Ploeger [GP09], and others [GP09, Mei89, Rei09], 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.

Outline of the paper. The remainder of the paper is structured as follows. In Section 2, we formally introduce reachability switching games (RSGs). In Section 3, we deal with one-player RSGs and show that solving them is `NP`-complete. We end this section by analysing the memory requirements of winning strategies in one-player RSGs. In Section 4, we deal with two-player RSGs, first showing that they can be solved in `EXPTIME` and then showing a `PSPACE` lower bound. We end this section by analysing the memory requirements of winning strategies in two-player RSGs. In Section 5, we deal with zero-player RSGs, first showing that solving them is `NL`-hard for the explicit representation and `P`-hard for the succinct representation, and then showing an upper bound of `UEOPL`. Finally, in Section 6, we discuss open problems and further work.

2. Preliminaries

In this section, we define *two-player* reachability switching games between a reachability player and a safety player. One-player games refer to games in which the safety player has no vertices, and zero-player games refer to games in which both the reachability and safety players have no vertices. The zero-player case corresponds to the problem studied by Dohrau et. al. [DGK+17], which they call Arrival.

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 $\mathrm{Ord}(v) = \langle u_1, u_2, \ldots, u_k \rangle$ 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 $(v_1, C_1), (v_2, C_2), \ldots$ 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 $v_i \in V_{\mathrm{Swi}}$ then $(v_{i+1}, C_{i+1}) = \mathrm{Succ}(v_i, C_i)$;
(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. For a zero-player game there is a single play, which is called a "run profile" by Dohrau et al. [DGK+17].

A (deterministic, history dependent) *strategy for the reachability player* is a function that maps each play prefix $(v_1, C_1), (v_2, C_2), \ldots, (v_k, C_k)$, with $v_k \in V_{\mathrm{R}}$, to an outgoing edge of v_k. A play $(v_1, C_1), (v_2, C_2), \ldots$ is *consistent* with a strategy if, whenever $v_i \in V_{\mathrm{R}}$, 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 $\mathrm{Ord}(v) = \langle u_1, u_2, \ldots, u_k \rangle$ gives a sequence of outgoing edges for every switching vertex. We consider two possible ways of representing $\mathrm{Ord}(v)$ in this paper. In *explicit* RSGs, $\mathrm{Ord}(v)$ is represented by simply writing down the sequence $\langle u_1, u_2, \ldots, u_k \rangle$.

Natural switching orders may contain runs of identical next nodes, e.g., $\mathrm{Ord}(v)$ may contain $u_\ell, u_{\ell+1}, \ldots, u_{\ell+m}$ with $u_\ell = u_{\ell+1} = \cdots = u_{\ell+m}$. In that case, if m is large the explicit representation is wasteful. Other, less wasteful representations are possible which give games with a different computational complexity to explicitly represented games. Motivated by this, we also consider games in which $\mathrm{Ord}(v)$ is written down in a more concise way, which we call *succinct* RSGs. In succinct games, for each switching vertex v, we have a sequence of pairs $\langle (u_1, t_1), (u_2, t_2), \ldots, (u_k, t_k) \rangle$, 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 $\mathrm{Ord}(v)$ should contain t_1 copies of u_1, followed by t_2 copies of u_2, and so on. So, if $\mathrm{Rep}(u, t)$ gives the sequence containing t copies of u, and if $\cdot$ represents sequence concatenation, then

$$\mathrm{Ord}(v) = \mathrm{Rep}(u_1, t_1) \cdot \mathrm{Rep}(u_2, t_2) \cdot \ \ldots \ \cdot \mathrm{Rep}(u_k, t_k).$$

Any explicit game can be written down in the succinct encoding by setting all $t_i = 1$. Note, however, that in a succinct game $\mathrm{Ord}(v)$ 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 $V_{\mathrm{S}} = \emptyset$.

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. [DGK+17] to show containment of the zero-player reachability problem in $\mathtt{NP} \cap \mathtt{coNP}$.

Overview. A controlled switching flow assigns an integer to every edge of the game. The flow is required to satisfy *balance* constraints that ensure that the flow entering a vertex is equal to the flow leaving that vertex (except for the source and target nodes). It is also required to satisfy a *switching* constraint, which will ensure that switching nodes send the correct amount of flow (i.e., consistent with the switching order) to each of their successors.

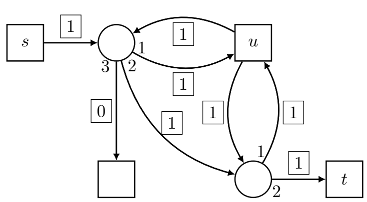


Figure 1: Three examples of controlled switching flows for the same game. The flows are given inside squares on the edges; the other numbers at the start of switching edges indicate the order of these switching edges. The left and centre examples correspond to plays, the rightmost example does not.

Intuitively, one might think of a switching flow as counting the number of times that an edge is used during a play. Indeed, if we take a winning play for the reachability player, and construct a flow by counting the number of times that each edge is used, then we will obtain a switching flow. The leftmost two diagrams in Figure 1 show two possible controlled switching flows for a game. In our diagramming notation, vertices controlled by the reachability player are represented as squares, while switching nodes are represented as circles. The switching order is shown by the numbers attached to the outgoing edges of the switching nodes.

The left hand diagram shows a controlled switching flow that corresponds to the player using the downwards edge from u once, and the leftwards edge from u once. The middle diagram shows a controlled switching flow that corresponds to the player using the downwards edge twice. Note that both strategies will result in the play reaching t.

Note that in the left hand diagram the order in which the edges are used is irrelevant: the player will win no matter which edge is used first. We will show that this is always the case. Given a controlled switching flow, the player can adopt a *marginal* strategy that simply uses each edge the proscribed number of times, and this is sufficient to ensure that the player will win.

One interesting feature of switching flows, pointed out by Dohrau et al. [DGK+17], is that there can exist *false* switching flows. These flows *do not* correspond to actual plays of the game. The rightmost diagram in Figure 1 shows a false controlled switching flow. Note that no possible play of the game uses the downwards edge three times while using the leftwards edge zero times, and so there is one unit of "false flow" between u and the vertex below it.

This does not affect our results in this section, since if the player follows a marginal strategy they will still win, though they will not be able to use the edge three times. However, we will have to deal with false flows when we show UEOPL containment for zero-player succinct games in Section 5.3.

Controlled switching flows. Formally, 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 $\mathrm{Bal}(F, v) = \sum_{(v,u)\in E} F(v, u) - \sum_{(w,v)\in E} F(w, v)$, which is the difference between the outgoing and incoming flow at v. For each switching node $v \in V_{\mathrm{Swi}}$, let $\mathrm{Succ}(v)$ denote the set of vertices that appear in $\mathrm{Ord}(v)$, and for each index $i \leq |\mathrm{Ord}(v)|$ and each vertex $u \in \mathrm{Succ}(v)$, let $\mathrm{Out}(v, i, u)$ be the number of times that u appears in the first i entries of $\mathrm{Ord}(v)$. In other words, $\mathrm{Out}(v, i, u)$ 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:

(1) The source vertex s satisfies $\mathrm{Bal}(F, s) = 1$, and the target vertex t satisfies $\mathrm{Bal}(F, t) = -1$.
(2) Every vertex v other than s or t satisfies $\mathrm{Bal}(F, v) = 0$.
(3) Let $v \in V_{\mathrm{Swi}}$ be a switching node, $k = |\mathrm{Ord}(v)|$, 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 $w \in \mathrm{Succ}(v)$ we have that $F(v, w) = p \cdot \mathrm{Out}(v, k, w) + \mathrm{Out}(v, q, w)$.

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

Proof. The proof will be by induction on the size of F. The base case is when $\sum_{e \in E} F(e) = 1$. The requirements of a controlled switching flow imply that $F(s,t) = 1$, and all other edges have no flow at all. If $s \in V_{\mathrm{R}}$, then the corresponding marginal strategy is required to choose the edge (s,t), and thus it is a winning strategy. If $s \in V_{\mathrm{Swi}}$, then the balance requirement of a controlled switching flow ensures that t is the first vertex in $\mathrm{Ord}(s)$, so the switching node will move to t, and the reachability player will win the game.

There are two cases to consider for the inductive step. First, assume that $\sum_{e \in E} F(e) = i$, and that $s \in V_{\mathrm{R}}$. Let (s,v) be the outgoing edge chosen by the marginal strategy (this can be any node that satisfies $F(s,v) > 0$). If G denotes the current game, then we can create a new switching game G', which is identical to G, but where v is the designated starting node. Moreover, we can create a controlled switching flow F' for G' by setting $F'(s,v) = F(s,v) - 1$ and leaving all other flow values unchanged. Observe that all properties of a controlled switching flow continue to hold for F'. Since $\sum_{e \in E} F'(e) = i - 1$, the inductive hypothesis implies that the marginal strategy that corresponds to F' (which is consistent with the marginal strategy for F) is winning for the reachability player.

The second case for the inductive step is when $\sum_{e \in E} F(e) = i$ and $s \in V_{\mathrm{Swi}}$. Let (s,v) be the first edge in $\mathrm{Ord}(s)$, which is the edge that the switching node will use. Again we can define a new game G' where the starting node is v, and in which $\mathrm{Ord}(s)$ has been rotated so that v appears at the end of the sequence. We can define a controlled switching flow F' for G' where $F'(s,v) = F(s,v) - 1$ and all other flow values are unchanged. Observe that F' satisfies all conditions of a controlled switching flow, and in particular that rotating $\mathrm{Ord}(s)$ allows s to continue to satisfy the balance constraint on its outgoing edges. Again, since $\sum_{e \in E} F'(e) = i - 1$, the marginal strategy corresponding to F' (which is identical to the marginal strategy for F) is winning for the reachability player. □

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

Lemma 3.2. *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 $n \cdot \ell^n$, where n is the number of nodes in the game and $\ell = \max_{v \in V_{Swi}} |\mathrm{Ord}(v)|$.*

Proof. Let $v_1, v_2, \ldots, v_k$ be the play that is produced when the reachability player uses his winning strategy. Define a flow F so that $F(e)$ is the number of times e is used by the play. We claim that F is a controlled switching flow. In particular, since the play is a path through the graph starting at s and ending at t, we will have $\mathrm{Bal}(F,s) = 1$ and $\mathrm{Bal}(F,t) = -1$, and we will have $\mathrm{Bal}(F,v) = 0$ for every vertex v other than s and t. Moreover, it is not difficult to verify that the balance constraint will be satisfied for every vertex $v \in V_{\mathrm{Swi}}$.

We now prove a bound on the size of the flow. First, observe that if a state (v,C) appears twice in the play, then we can modify the strategy to eliminate this cycle. Since the strategy is winning, we know that the original play must be finite, and so we can apply the previous argument finitely many times to produce a winning strategy that visits each state at most once. Since the size of F is equal to the number of steps in the play, we can upper bound the size of F by the number of distinct states. Recall that C consists of $|V_{\mathrm{Swi}}|$ numbers, and that $C(v)$ can take at most $|\operatorname{Ord}(v)|$ different values. So the number of possible values for C is at most $\ell^{|V_{\mathrm{Swi}}|}$, and so the number of possible states is at most $|V| \cdot \ell^{|V_{\mathrm{Swi}}|} \leq n \cdot \ell^n$. □

Combing the two previous lemmas yields the following corollary.

Corollary 3.3. *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 3.4. *Deciding the winner of an explicit or succinct one-player RSG is in* NP.

Proof. By Lemma 3.1 and Lemma 3.2, the reachability player can win if and only if the game has a controlled switching flow of size at most $n \cdot \ell^n$. So, we can non-deterministically guess a flow of size $n \cdot \ell^n$ and then verify that it satisfies the requirements of a controlled switching flow. For explicit games (where $\ell \in O(n)$) this can clearly be done in polynomial time.

For succinct games, first observe that, if N denotes the input size of the game, then $\ell \leq 2^N$. Thus, the size of the flow is at most $n \cdot 2^{N \cdot n}$. Since the flow is represented by a set of numbers, each of which is written in binary, it can be represented by at most $\log(n \cdot 2^{N \cdot n})$ bits, which is polynomial in the input size.

Secondly, we note that the requirements of a switching flow can still be checked in polynomial time, even for a succinct RSG. The first two requirements give balance constraints that do not refer to the switching order, and thus can be checked in polynomial time no matter whether the game is explicit or succinct.

The third requirement of a controlled switching flow does refer to the switching order. It requires us to check, for each switching node v, that the correct amount of flow is sent along each edge (v, u), and the condition is written in terms of $\operatorname{Out}(v, i, u)$. Recall that $\operatorname{Out}(v, i, u)$ denotes the number of times that u appears in the first i entries of $\operatorname{Ord}(v)$, and observe that this can be computed in polynomial time, even when $\operatorname{Ord}(v)$ is given succinctly. Thus, the third constraint of a controlled switching flow can also be checked in polynomial time for succinct games. □

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 n variables x_1, x_2, …, x_n, and m clauses C_1, C_2, …, C_m. Thus, the overall size of the 3SAT formula is $n + m$. It is well-known [Tov84, 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 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 Figure 2.

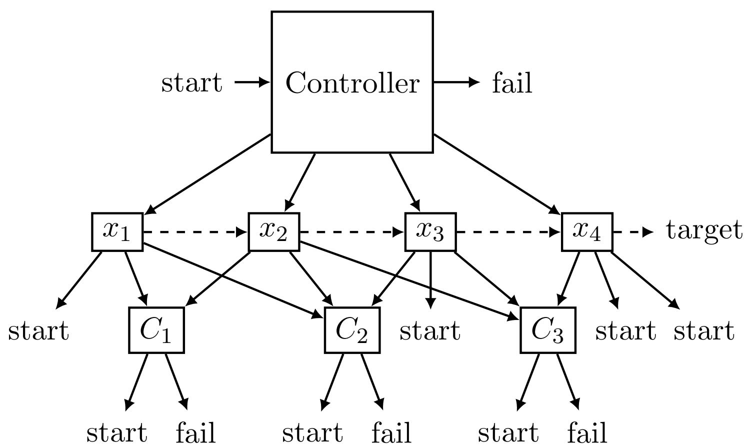


Figure 2: Overview of our construction for one player for the example formula $C_1 \wedge C_2 \wedge C_3 = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_3 \vee x_4)$. Note that the negations of variables in the formula are not relevant for this high-level view; they 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.

The control gadget. The sequencing in the construction is determined by the control gadget, which is shown in Figure 3. Recall that, 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 Figure 3 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

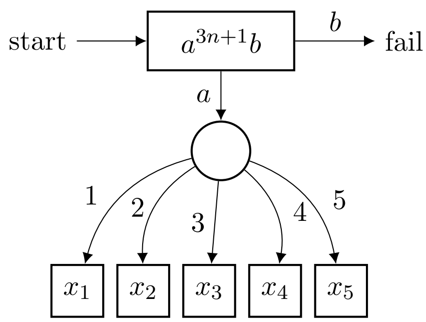


Figure 3: The control gadget.

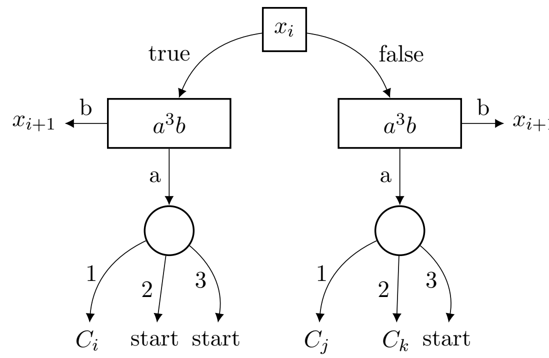


Figure 4: A variable gadget.

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. 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 Figure 4. 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 Figure 5. The gadget counts how many variables have failed to satisfy the

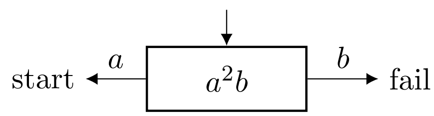


Figure 5: A gadget for a clause with three variables.

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

We split the two directions into two separate lemmas.

Lemma 3.6. *If there is a satisfying assignment to the 3SAT formula, then the reachability player can win the one-player RSG.*

Proof. The strategy for the reachability player is as follows: at each variable vertex x_i, choose the branch that corresponds to the value of x_i in the satisfying assignment. We argue that this is a winning strategy. First note that the game cannot be lost in a clause gadget during the variable phase. Since the assignment is satisfying, the play cannot visit a clause gadget more than twice (or more than once if the clause only has two variables), and therefore the edges from the counting gadgets to the fail vertex cannot be used. Hence, the game will eventually reach the verification phase. At this point, since the strategy always chooses the same branch, the play will pass through x_1, x_2, …, x_n, and then arrive at x_{n+1}. Since this is the target, the reachability player wins the game. □

Lemma 3.7. *If the reachability player wins the one-player RSG, then there is a satisfying assignment of the 3SAT formula.*

Proof. We begin by arguing that, if the reachability player wins the game, then he must have chosen the same branch at every visit to every variable gadget. This holds because x_{n+1} can only be reached by ensuring that each variable has a branch that is visited at least 3 times. The control gadget causes the reachability player to immediately lose the game if it is visited $3n + 2$ times. Thus, the reachability player must win the game after passing through the control gadget exactly $3n + 1$ times. The only way to do this is to ensure that each variable has a branch that is visited exactly 3 times during the variable phase.

Thus, given a winning strategy for the game, we can extract a consistent assignment to the variables in the 3SAT instance. Since the game was won, we know that the game did not end in a clause gadget, and therefore under this assignment every clause has at least one literal that is true. Thus, the assignment satisfies the 3SAT instance. □

Note that our game can be written down as an explicit game whose size is polynomial in $n + m$, so our lower bound applies to both explicit and succinct games. Hence, we have the following theorem.

Theorem 3.8. *Deciding the winner of an explicit or succinct one-player RSG is* NP*-hard.*

3.3. **Memory requirements of winning strategies in one-player games.** In this section we consider the memory requirements for winning strategies in one-player reachability switching games. For general history-dependent strategies, where the player has access to the current switch configuration, there exist winning strategies that use no memory (this will be argued formally in the proof of Lemma 4.6). Here we consider the scenario where the player *does not* have access to the current switch configuration, but *does* have access to the current vertex and to some finite memory. We will show an exponential lower bound on the amount of memory that the player needs to execute a winning strategy.

Consider the game shown in Figure 6, which takes as input a parameter p that we will fix later.

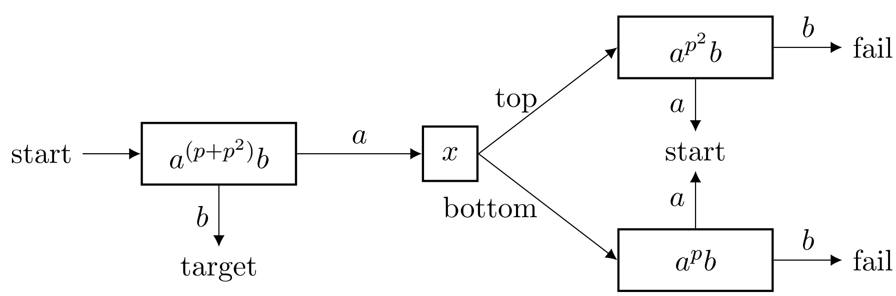


Figure 6: One-player memory lower bound construction.

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 3.9. *The reachability player must use at least $p - 1$ memory states to win the game shown in Figure 6.*

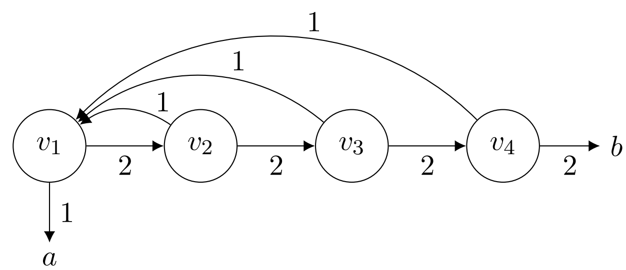


Figure 7: A gadget that produces $a^{15}b$.

Proof. Consider a winning strategy and let M denote the number of memory states that it requires. Run the strategy until the vertex x is visited for the pth time, and keep track of the memory states that are visited by the strategy. At this point there are two possibilities. If no memory state has been visited twice, then the strategy has used $p-1$ distinct memory states, and so the claim has been shown.

Alternatively, if the strategy has used the same memory state twice, then there must be a cycle of memory states, and since the player has only one vertex, we know that this cycle will be repeated until the end of the game. We have the following facts about this cycle:

- The bottom edge was used used at most $p-1$ times before the cycle started, but all winning strategies must use the bottom edge p times. Therefore the cycle must use the bottom edge at least once or the player will lose.
- Since the bottom edge cannot be used more than p times, and each iteration of the cycle uses the bottom edge at least once, it follows that the cycle cannot be repeated more than p times.
- The top edge was used at most $p-1$ times before the cycle started, and so it must be visited at least $p^2-(p-1)$ times before the game is won.

From the above, we get that each iteration of the cycle must use the top edge $O(p)$ many times, since otherwise the last two constraints above could not be satisfied. This means that the cycle uses $O(p)$ memory states.

To make the argument above more precise, let C_T and C_B denote the number of times the strategy chose top and bottom, respectively, in the prefix before we entered the cycle of memory states. Let L denote the length of the cycle of memory states, and L_T denote the number of memory states on the cycle where the strategy chooses top. The cycle must use the top edge p^2-C_T times, and the number of times that the cycle can repeat is bounded by $p-C_B$. So, we get the following bound on the number of memory states:

$$M \geq L \geq L_T \geq \frac{p^2-C_T}{p-C_B} \geq \frac{p^2-p}{p} = p-1,$$

which completes the proof. $\square$

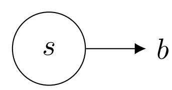
We will set $p = 2^{n/2}$ to obtain our lower bound. We show that, even though p is exponential, it is possible to create an explicit switching gadget that produces the sequence $a^{2^n}b$ using n switching nodes.

To show this, we will build gadgets that produce the sequence $a^x b$ where x is a number encoded in binary. The construction is given in the following lemma. An example gadget produced by the construction is given in Figure 7.

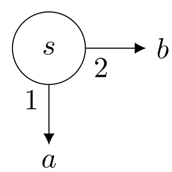
Lemma 3.10. *For all $x \in \mathbb{N}$ there is an explicit switching gadget of size $\log_2(x)$ with output $a^x b$.*

Proof. We will build up the construction recursively. Each gadget will have a start state s, and two output states a and b. Each time the token enters the gadget at s, it leaves via a or b. We are interested in sequence of outputs generated if the token repeatedly arrives at s. Given a word w over the alphabet $\{a, b\}$ (eg. $abba$) we say that a gadget produces that word if repeatedly feeding the token through the gadget produces the sequence w on the outputs. For every x, we will denote the gadget that outputs the word $a^x b$ as GADGET($a^x b$).

For the base case of the recursion, we consider all $x \leq 2^0 = 1$. For $x = 0$ we use the following gadget:

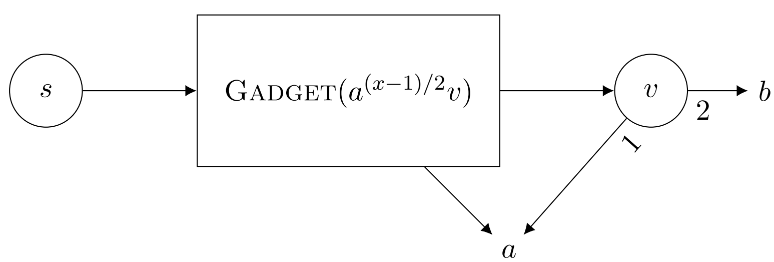


For $x = 1$ we use the following gadget:



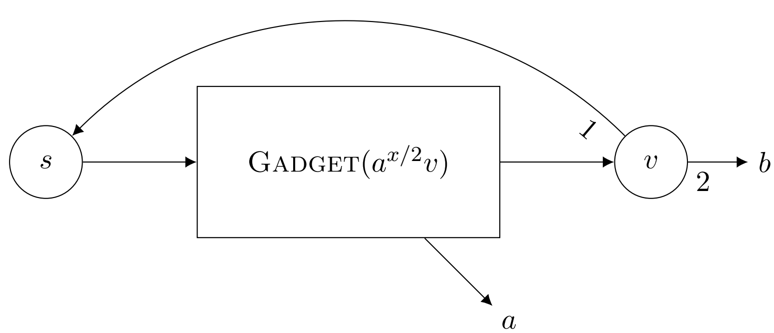
The correctness of both of these gadgets is self-evident.

Now, suppose that we have gadgets for all $x' \leq 2^{i-1}$, and let x be a number with $2^{i-1} < x \leq 2^i$. If x is odd, then we use the following construction:



The gadget produces the sequence $a^{(x-1)/2} \cdot a \cdot a^{(x-1)/2} \cdot b = a^x b$, where $\cdot$ denotes concatenation.

If x is even, then we use the following gadget.



The gadget produces $a^{x/2} \cdot a^{x/2} \cdot b = a^x b$.

So, we have provided a family of gadgets the produce sequences of the form $a^x b$. Since each iteration of the recursion divides x by two, and since each iteration adds at most one new state, we have that $\textsc{Gadget}(a^x b)$ uses $\log_2(x)$ switching nodes. □

Theorem 3.11. *The number of memory states needed in an explicit one-player reachability switching game is $2^{\Omega(n)}$, where n is the number of states.*

Proof. If we set $p = 2^{k/2}$, and use the gadgets from Lemma 3.10, then the game in Figure 6 has $n = O(\log(p^2)) = O(\log(2^n)) = O(k)$ states. Lemma 3.9 shows that the reachability player needs $p - 1 = 2^{k/2} - 1$ memory states to win the game. Hence, $2^{\Omega(n)}$ memory states are required. □

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 4.1. *Deciding the winner of an RSG is in* `EXPTIME`.

Proof. We prove this by showing that the game can be simulated by an *alternating Turing machine*, which is a machine that has both existential and universal non-determinism. It has been shown that `APSPACE` = `EXPTIME` [CKS81], which means that if we can devise an algorithm that runs in polynomial space on an alternating Turing machine, then we can obtain an algorithm that runs in exponential time on a deterministic Turing machine.

It is straightforward to implement an explicit or succinct RSG on an alternating Turing machine. The machine simulates a run of the game. It starts by placing a token on the starting state. It then simulates each step of the game. When the token arrives at a vertex belonging to the reachability player, it uses existential non-determinism to choose a move for that player. When the token arrives at a vertex belonging to the safety player, it uses universal non-determinism to choose a move for that player. The moves at the switching nodes are simulated by remembering the current switch configuration, which can be done in polynomial space. The machine accepts if and only if the game arrives at the target state.

This machine uses polynomial space, because it needs to remember the switch configuration. Note that it still uses polynomial space even for succinct games, since a switch configuration for a succinct game can be written down as a list of n numbers expressed in binary. This completes the proof of Theorem 4.1. □

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.

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 Figure 8.

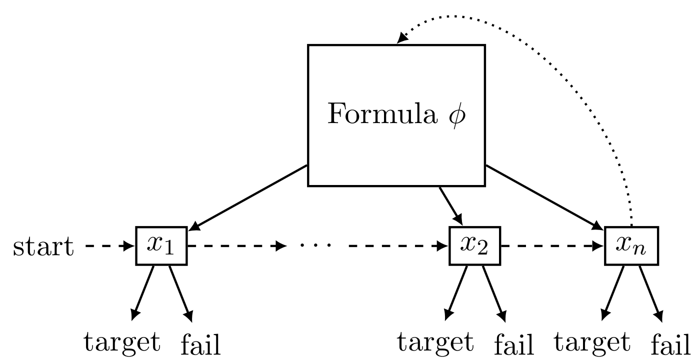


Figure 8: High-level overview of our construction for two players. The dashed lines between variables are part of the first, quantifier phase; the dotted line from variable x_n to the Formula is the transition between phases, and the solid edges are part of the second, formula phase.

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 Figure 9.

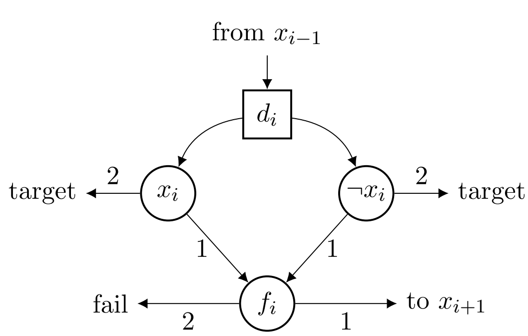


Figure 9: The initialization gadget for an existentially quantified variable x_i.

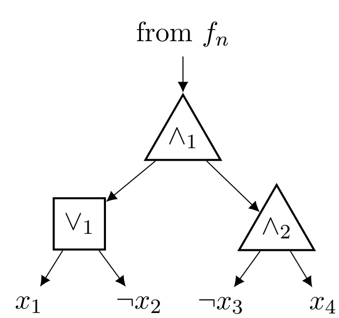


Figure 10: The formula phase game for the formula $(x_1 \vee \neg x_2) \wedge \neg x_3 \wedge x_4$.

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.

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 Figure 10. In our diagramming notation, nodes controlled by the safety player are represented by triangles.

Intuitively, if ϕ is satisfied by the assignment to $x_1, \ldots, x_n$, 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 4.2. *The reachability player wins the RSG if and only if the QBF formula is true.*

Proof. If the QBF formula is true, then during the quantifier phase, no matter what assignments the safety player picks for the universally quantified variables, the reachability player can choose values for the existentially quantified variables in order to make ϕ true. Then, in the formula phase the reachability player has a strategy to ensure that he wins the game, by moving to a node x_i or $\neg x_i$ that was used during the quantifier phase.

Conversely, and symmetrically, if the QBF formula is false then the safety player can ensure that the assignment does not satisfy ϕ during the quantifier phase, and then ensure that the game moves to a node x_i or $\neg x_i$ that was not used during the quantifier phase. This ensures that the safety player wins the game. $\square$

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 4.3. *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 4.4. *Deciding the winner of a polynomial-length RSG is* `PSPACE`*-complete.*

Proof. Hardness follows from Theorem 4.3. For containment, observe that the simulation by an alternating Turing machine described in Section 4.1 runs in polynomial time whenever the game terminates after a polynomial number of steps. Hence, we can use the fact that $\mathtt{AP} = \mathtt{PSPACE}$ [CKS81] to obtain a deterministic polynomial space algorithm for solving the problem. □

4.3. **Memory requirements for two-player games.** As in Section 3.3, we now consider the scenario in which the player does not have access to the switch configuration, but does have access to a finite memory. We are able to show a stronger memory lower bound for two-player games compared to one-player games. Figure 11 shows a simple gadget that forces the reachability player to use memory. The game starts by allowing the safety player

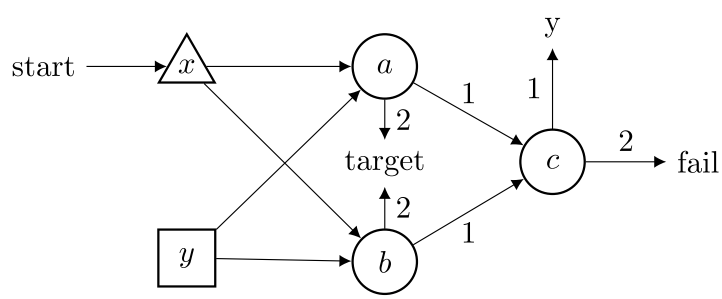


Figure 11: An RSG in which the reachability player needs to use memory.

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 k copies of the memory gadget, we can show the following lower bound. Observe that, in contrast to Theorem 3.11, no asymptotics are used in this lower bound.

Lemma 4.5. *In an explicit or succinct RSG, winning strategies for both players may need to use $2^{n/6}$ memory states, where n is the number of states.*

Proof. Consider a game with k copies of the memory gadget shown in Figure 11, but modified so that the following sequence of events occurs:

(1) The safety player selects a or b in all gadgets, one at a time.
(2) The safety player then moves the game to one of the y vertices in one of the gadgets.
(3) The reachability player selects a or b as normal, and then either wins or loses the game.

Each gadget has five states giving $5 \cdot k$ states in total, and one extra state is needed for the safety player to make the decision in Step 2. Hence, the total number of states is $n = 5 \cdot k + 1 \leq 6 \cdot k$, since we can assume that $k \geq 1$.

The reachability player has an obvious winning strategy in this game, which is to remember the choices that the safety player made, and choose the same vertex in the third step. As the safety player makes k binary decisions, this strategy uses 2^k memory states.

On the other hand, if the reachability player uses a strategy σ with strictly less than 2^k memory states, then the safety player can win the game in the following way. There are 2^k different switch configurations that the safety player can create at the end of the first step of the game. By the pigeon-hole principle there exists two distinct configurations C_1 and C_2 that are mapped to the same memory state by σ. The safety player selects a gadget i that differs between C_1 and C_2, and determines whether σ selects a or b for gadget i. He then selects the configuration that that is consistent with the other option, so if σ chooses a the safety player chooses the configuration C_j that selects b. He then sets the gadgets according to C_j in step 1, and moves the game to gadget i in step 2. The reachability player will then select the vertex not chosen in step 1, so he loses the game.

Hence, the reachability player must use 2^k memory states to win the game. Since the game has $n \leq 6 \cdot k$ states, this gives a lower bound of $2^{n/6}$ memory states. Finally, observe that we can obtain the same lower bound for the safety player by swapping the roles of both players in this game. □

Straightforward upper bound. It should be noted that that exponential memory is sufficient in a two-player reachability switching game. We say that a strategy is a *switch configuration strategy* if it simply remembers the current switch configuration. Any such strategy uses at most exponentially many memory states. For games with binary switch nodes, these strategies use exactly 2^n memory states, where n is the number of switching nodes.

Lemma 4.6. *In a reachability switching game, both players have winning switch configuration strategies.*

Proof. Let $G = (V, E, V_{\mathrm{R}}, V_{\mathrm{S}}, V_{\mathrm{Swi}}, o, s, t)$ be a reachability switching game, and let $\mathcal{C}$ denote the set of all switch configurations in this game. Consider the "blown-up" reachability game G' played on $V \times \mathcal{C}$, where there are no switching nodes, but instead the successor of a vertex (v, C) with $v \in V_{\mathrm{Swi}}$ is determined by C. It is straightforward to show that the reachability player wins the game G' if and only if he wins the original game. Both players in a reachability game have positional winning strategies. Therefore, if a player can win in G', then he can also win in G using a switch configuration strategy that always plays according to the positional winning strategy in G'. □

5. Zero-player reachability switching games

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

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 $|V|$, as follows:

- The start vertex is $(s, 1)$.
- The target vertex is fin.
- For all k, the vertex (t, k) has a single outgoing edge to fin. This means that if we ever reach t then we win the game.
- Each vertex (v, k) with $v \neq t$ and $k < |V|$ has the following outgoing edges.
  - If v has at least one outgoing edge, then (v, k) has an edge to $(u, k+1)$, for each edge $(v, u) \in E$.
  - If v has no outgoing edges, then (v, k) has a single outgoing edge to $(s, 1)$.

  In other words, the switching node (v, k) cycles between the outgoing edges of v, and increases the count by 1, or it goes back to the start vertex if v is a dead-end.
- Each vertex $(v, |V|)$ with $v \neq t$ has a single outgoing edge to $(s, 1)$. This means that when the count reaches $|V|$ the we start again from $(s, 1)$.

This game can be constructed in logarithmic space by looping over each element in $V \times V \cup \{\mathsf{fin}\}$ and producing the correct outgoing edges.

Theorem 5.1. *Deciding the winner of an explicit zero-player RSG is* NL*-hard under logspace reductions.*

Proof. We must argue that there is a path from s to t if and only if the zero-player reachability game eventually arrives at fin. By definition, if the game arrives at fin, then there must be a path from s to t, since all paths from $(s, 1)$ to a vertex (t, k) only use edges from the original graph.

For the other direction, suppose that there is a path from s to t, but the game never arrives at fin. By construction, if the game does not reach fin, then $(s, 1)$ is visited infinitely often. Since $(s, 1)$ is a switching state, we can then argue that the vertex $(v, 2)$ is visited infinitely often for every successor v of s. Carrying on this argument inductively allows us to conclude that if there is a path of length k from s to v, then the vertex (v, k) is visited infinitely often, which provides our contradiction. □

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 [DGK+17]. The fact that the problem lies in NP follows from Theorem 3.4, 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. [DGK+17, 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. [GHR95]. 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.

Figure 12 shows our construction for an AND-gate of depth 2. It consists of a single

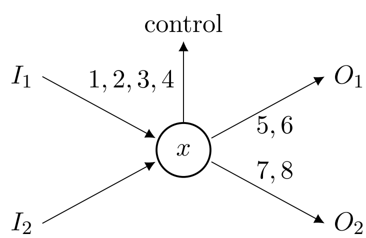


Figure 12: An AND-gate of depth 2.

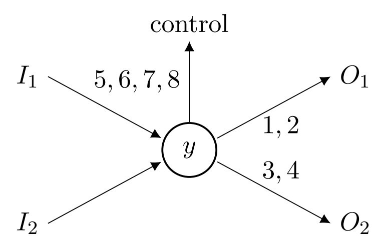


Figure 13: An OR-gate of depth 2.

switching node (with a succinct order). Further, 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. Figure 13 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. The purpose of the control state is to send the correct signals to the input gates. Specifically, the input bits that are true should receive 2^d signals, where d is the depth of the circuit, while the input bits that are false should receive no signals.

This is achieved in the following way. 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 5.2. *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.*

Proof. We prove the two directions separately.

The $\Rightarrow$ direction. Here we must show that if the game reaches the target state, then gate g evaluates to true on the input bit-string. We prove the following two statements by induction:

(1) Every gate at depth i receives at most 2^i signals from each of its inputs.
(2) For all gates g' in the circuit, if g' sends at least one signal to one of its outputs, then g' evaluates to true when the circuit is evaluated on B.

Note that this is sufficient to prove the claim, since point (2) implies that if gate g sends the token to the target, then g must evaluate to true when the circuit is evaluated on B.

For the base case, we use the signals generated by the control state. Note that the control state produces either 2^d inputs or 0 inputs for each input line, which proves point (1), and the control state sends signals to a gate of depth d if and only if the corresponding bit of B is true, which proves point (2).

For the inductive step, let g' be a gate at depth i, and suppose that g' is an AND-gate. To prove point (1) note that, by the inductive hypothesis, the gate g' can receive at most 2^i input signals from each input, giving a total of 2^{i+1} input signals in total. No matter whether g' is an AND-gate or an OR-gate, our gadgets ensure that at most 2^{i-1} signals can be sent to each output, and any remaining signals will be sent to the control state.

For point (2), let us first assume that g' is an AND-gate. If g' sends a signal to one of its outputs, then it must have received strictly more than 2^i input signals. Point (1) tells us that the only way this is possible is if both of input gates sent signals to g'. Thus, by the inductive hypothesis, both of the inputs to g' evaluate to true when the circuit is evaluated on B, and therefore g' must also evaluate to true when the circuit is evaluated on B.

Note suppose that g' is an OR-gate. By construction, if g' sends a signal to one of its outputs, then at least one of the inputs to g' must have sent a signal to g'. By point (2) of the inductive hypothesis, this means that at least one input of g' evaluates to true when the circuit is evaluated on B. This means that g' must also evaluate to true when the circuit is evaluated on B.

The $\Leftarrow$ direction. We show that if gate g evaluates to true on the bit-string B, then the target state will be reached. So, suppose for the sake of contradiction, that g evaluates to true, but the target state was not reached. Note that the game cannot continue indefinitely, because the control state appears on every cycle of the game, and eventually the control state will send the token to the absorbing failure state. So, since the target was not reached, this means that the token must have arrived at the failure state.

Since the token arrived at the failure state, this means that the gates at depth d received the correct input signals for the bit-string B. By construction, this means that they outputted correct signals to the gates at depth $d-1$. Applying this reasoning inductively, we can conclude that the gate g received correct input signals from its inputs. But, since gate g evaluates to true, this means that it sent the token to the absorbing target state, which contradicts the fact that the token arrived at the failure state. □

Since these gadgets use exponentially large 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 5.3. *Deciding the winner of a succinct zero-player RSG is* `P`*-hard under logspace reductions.*

5.3. **Succinct Zero-Player Games are in UEOPL and CLS.**

Gärtner et al. [GHH+18] have shown that the problem of solving an explicit zero-player game lies in `CLS` (which has recently been shown to be equal to `PPAD` $\cap$ `PLS` [FGHS21]). Their proof reduces the problem to End-of-Metered-Line, which is a problem that lies in `CLS` [HY17]. End-of-Metered-Line has also been shown to lie in the recently defined complexity class `UEOPL` [FGMS20].

In this section, we show that *succinct* zero-player games also lie in both `CLS` and `UEOPL`. We do so by adopting the same strategy as Gärtner et al., namely reducing to End-of-Metered-Line.

True and false switching flows. The crux of the reduction of Gärtner et al. is a method for differentiating between *true* and *false* switching flows. A switching flow for a zero-player game is simply the specialization of a controlled switching flow, which we defined in Section 3, in which all nodes are switching nodes. This matches the original definition of a switching flow given by Dohrau et al. [DGK+16].

In this section, it will be convenient to consider switching flows for which the final node is not necessarily the target of the game. We say that a switching flow *has target* x, if it is a switching flow for a game in which the target node is x.

Since a switching flow is just a specialization of a controlled switching flow, Lemma 3.1 and Lemma 3.2 already prove that the reachability player wins if and only if there is a switching flow, which was already observed by Dohrau et al. [DGK+16]. However, they point out that not every switching flow corresponds to an actual outcome of the game.

Since we are in the zero-player setting, there is exactly one play of the game. For each integer i, let $N_i : E \to \mathbb{N} \cup \{\infty\}$ be the function that gives the number of times each edge is used by the first i steps of the play. We call these functions the *run profiles* of G.

We define x_i to be the last vertex visited by the first i steps of the play. It is not difficult to prove that each N_i is a switching flow with target vertex x_i, but the converse is not always true. Specifically, there can exist switching flows F such that for all i we have $F(e) \neq N_i(e)$ for at least one edge e.

For example, consider the right-hand diagram in Figure 1. There are three units of flow leaving u, but if the player follows the marginal strategy associated with the flow, then the downwards edge from u will only be used twice. Thus, there is one unit of "false flow" between u and the vertex immediately beneath it. While the diagram depicts a one-player game, the phenomenon can also occur in the zero-player setting, as pointed out by Dohrau et al. [DGK+16].

This leads us to the following definition.

Definition 5.4 (True/False Switching Flows)**.** A switching flow $F : E \to \mathbb{N}$ is said to be true if there exists and i such that $F(e) = N_i(e)$ for all edges $e \in E$, and it is said to be false if this is not the case.

Note that false switching flows still witness reachability, but they do not correctly characterise a run profile of the game.

Detecting false switching flows. Gärtner et al. give the following characterisation of true switching flows. Their work considered *binary* explicit games, meaning that every vertex has exactly two outgoing edges.

Given a switching flow F in a binary game, one can easily determine the *most recently used edge* at each vertex.

Definition 5.5 (Most Recently Used Edge – Binary Game)**.** Suppose that vertex v has two outgoing edges (v, u_1) and (v, u_2), and that the switching order for v is $\langle u_1, u_2 \rangle$.

- if $F(v, u_1) = F(v, u_2) > 0$ then the most recently used edge is u_2,
- if $F(v, u_1) = F(v, u_2) + 1$ then the most recently used edge is u_1, and
- if $F(v, u_1) = F(v, u_2) = 0$ then there is no recently used edge.

Note that no other possibility is allowed by the definition of a switching flow.

The MRU graph of a switching flow F in a game G is denoted as $\mathrm{MRU}(G, F)$, and it is obtained by deleting all edges from G that are not a most recently used edge. Note that every vertex has at most one outgoing edge in an MRU graph. Gärtner et al. use the MRU graph to give the following characterisation of false switching flows.

Lemma 5.6 (Gärtner et al. [GHH+18])**.** *Let F be a switching flow for a binary explicit zero-player switching game with target x. Then F is a true switching flow if and only if one of the following conditions holds.*

(1) $\mathrm{MRU}(G, F)$ *is acyclic.*
(2) *There is exactly one cycle in* $\mathrm{MRU}(G, F)$*, and the target node x lies on this cycle.*

The idea here is that, if the MRU graph of F contains a cycle that does not contain the target node, then that cycle contains false flow. Specifically, for each edge e on that cycle, there will be no index i such that $F(e) = N_i(e)$. Conversely, the non-existence of such a cycle is sufficient to ensure that F is a true switching flow. We will extend this lemma to cover succinct non-binary zero-player games.

Generalising most-recently used edges. The definition of a most-recently used edge can be generalised to non-binary succinct games in a natural way.

Definition 5.7 (Most Recently Used Edge – Succinct Non-Binary-Game)**.** Let F be a switching flow, let v be a vertex, and let $\langle (u_1, t_1), (u_2, t_2), \dots, (u_m, t_m) \rangle$ be the switching order at v, which may be succinct or explicit. If $t = \sum_{(w,v) \in E} F(w, v)$ denotes the total amount of flow incoming at v, and $k = \sum_{i=1}^{m} t_i$ is the length of the switching order at v, then we use the following definitions.

- If $t = 0$, then there is no most recently used edge at v.
- If $t > 0$ and $t \bmod k = 0$, then the most recently used edge at v is u_m.
- If $t > 0$ and $t \bmod k = i$, then the most recently used edge at v is the vertex u that appears at position $i - 1$ in $\mathrm{Ord}(v)$.

Observe that, even for succinctly represented orderings, we can compute the most recently used edge at each vertex in polynomial time.

Recall that, given a flow F in a game G, the graph $\mathrm{MRU}(G, F)$ contains the set of most recently used edges in F. This definition also applies to non-binary succinct games, using the definition of a most-recently used edge given above.

Non-binary explicit games. We begin by slightly generalising Lemma 5.6 to non-binary explicit games. In fact, the proof of Gärtner et al. essentially already works for non-binary games, but several details need to be updated, and so we adapt the proof here ourselves for the sake of completeness.

Lemma 5.8. *Let F be a switching flow for an explicit zero-player switching game with target node x. Then F is a true switching flow if and only if one of the following conditions holds.*

(1) $\mathrm{MRU}(G, F)$ *is acyclic.*
(2) *There is exactly one cycle in* $\mathrm{MRU}(G, F)$*, and the target node x lies on this cycle.*

Proof. We prove the directions separately.

The $\Rightarrow$ direction: For the forward direction, we must show that if F is a true switching flow, then the MRU graph of F satisfies the conditions. Let i be the index such that $F = N_i$. Observe that the MRU graph of F is the same as the MRU graph of N_i. We will show that the conditions hold for N_i by induction on i.

For the base case, observe that N_0 corresponds to the prefix of the run that has not used any edges, and so the MRU graph of N_0 is empty, and thus acyclic. For the inductive step, suppose that the MRU graph of N_j satisfies the conditions, and that the $j + 1$th step of the play moves from v to u. This transforms the MRU graph in the following way: the existing edge at v is deleted, and the new most-recently used edge of v is set to (v, u). If the MRU graph of N_j contains a cycle, then by the inductive hypothesis it must pass through v. Therefore, deleting the outgoing edge of v makes the graph acyclic. Adding the edge (v, u) to the graph may introduce a new cycle, but this passes through u, which is the target vertex of N_{j+1}. Hence, the MRU graph of N_{j+1} satisfies the conditions.

To conclude the forward direction of the proof, we have shown that each run profile N_i satisfies the conditions of the lemma. Since $F = N_i$ for some i, this means that F also satisfies the conditions of the lemma.

The $\Leftarrow$ direction: For the other direction, we must show that if the conditions on the MRU graph are satisfied, then F is a true switching flow. For the sake of contradiction, we suppose that this is not the case. Let i be the largest index such that $N_i(e) \leq F(e)$ for all edges e, and let δ be defined so that $\delta(e) = F(e) - N_i(e)$ for all edges e.

Let x be the target node of $N_i(e)$. The first observation is that $\delta(x, u) = 0$ for all edges (x, u). To see why, observe that if $\sum_{(v,x)\in E} N_i(v, x) < \sum_{(v,x)\in E} F(v, x)$, ie., if x receives less total flow under N_i than it does under F, then the switching flow N_{i+1} will also satisfy $N_i(e) \leq F(e)$ for all edges e. On the other hand, if $\sum_{(v,x)\in E} N_i(v, x) = \sum_{(v,x)\in E} F(v, x)$, then the switching flow constraints force both flows to send the same amount of flow to each outgoing edge of x, which implies that $\delta(x, u) = 0$ for all edges (x, u).

If $\delta(e) = 0$ for all edges e, then we have a contradiction with the assumption that F is a false switching flow. Otherwise, we will prove that there is a cycle in the MRU graph of F that only uses edges e with $\delta(e) > 0$.

First note that if $\delta(v, u) > 0$ for some edge (v, u), then there must be an edge (u, w) satisfying $\delta(u, w) > 0$. This is because the flow constraints ensure that $\sum_{(x,u)\in E} F(x, u) = \sum_{(u,y)\in E} F(u, y)$ and also $\sum_{(x,u)\in E} N_i(x, u) = \sum_{(u,y)\in E} N_i(u, y)$. Since we have by assumption that $F(v, u) > N_i(v, u)$, and since $F(e) \geq N_i(e)$ for all edges e, there must be an outgoing edge at u satisfying $F(u, w) > N_i(u, w)$.

Next we show that if v has an outgoing edge e satisfying $\delta(e) > 0$, then if e' denotes the most-recently used edge at v, then we also have that $\delta(e') > 0$. Suppose that the switching order at v is $\langle u_1, u_2, \ldots, u_k\rangle$, and suppose that $\delta(v, u_j) > 0$ for some j. There are two cases to consider.

- If $F(v, u_1) = F(v, u_2) = \cdots = F(v, u_k)$, that is, if F sends the same amount of flow to all successors of v, then we must have $F(v, u_k) > N_i(v, u_k)$ due to the fact that $\delta(v, u_j) > 0$, and the fact that the switching flow constraints ensure that flow is sent to u_j before it is sent to u_k. Since the MRU graph uses u_k at v by definition, this implies that MRU graph uses an edge at v with $\delta(e) > 0$.
- If $F(v, u_1) = \cdots = F(v, u_\ell) = F(v, u_{\ell+1}) + 1 = \cdots = F(v, u_k)$, that is, if the flow sends one unit of flow more to vertices u_1 through u_ℓ, then observe the following.
  - If $j < \ell$ then as before, since $\delta(v, u_j) > 0$, and the fact that the flow constraints send flow to u_j before u_ℓ, we have $\delta(v, u_\ell) > 0$.
  - If $j = \ell$ then $\delta(v, u_\ell) = \delta(v, u_j) > 0$ by assumption.
  - If $j > \ell$ then since $N_i(u_j) \leq N_i(u_\ell)$, we have that
$$F(v, u_\ell) - N_i(v, u_\ell) \geq F(v, u_j) - N_i(v, u_j) > 0.$$

  The MRU graph uses u_ℓ at v by definition, and so we again have shown that the MRU graph uses an edge at v with $\delta(e) > 0$.

We can now prove that the MRU graph of F contains a cycle that that only uses edges e with $\delta(e) > 0$. We can construct this cycle in the following way.

(1) Start with an arbitrary edge (v, u) satisfying $\delta(v, u) > 0$.
(2) Identify the most recently used edge (v, w) at v. We have proved that $\delta(v, w) > 0$.
(3) We have shown that since $\delta(v, w) > 0$, there exists an outgoing edge (w, x) satisfying $\delta(w, x) > 0$.
(4) Repeat the argument starting from Step 2 with the edge (w, x).

This argument constructs a path of most-recently used edges such that each edge e satisfies $\delta(e) > 0$. Note that the path can never visit the target vertex, since we have shown that all outgoing edges e of the target satisfy $\delta(e) = 0$. Since the graph is finite, this path must eventually loop back to itself, which creates a cycle. Thus, the MRU graph contains a cycle that does not contain the target vertex, which gives a contradiction. □

Succinct games. We can now prove that the same property holds for succinct games. Note that none of the properties in Lemma 5.8 care about the size of the game. So, for each vertex v in a succinct game G, we can take a succinct ordering $\mathrm{Ord}(v) = \langle (u_1, t_1), (u_2, t_2), \ldots, (u_k, t_k) \rangle$, and blow it up to the explicit switching order

$$\mathrm{Rep}(u_1, t_1) \cdot \mathrm{Rep}(u_2, t_2) \cdot \ \ldots \ \cdot \mathrm{Rep}(u_k, t_k)$$

to obtain a (potentially exponentially large) explicit game G'. We can then apply Lemma 5.8 to G', and observe that an edge is most recently used in G' if and only if it is most recently used in G, since we have not actually changed the switching order at any vertex. So, we obtain the following:

Lemma 5.9. *Let F be a switching flow for a succinct zero-player switching game with target node x. Then F is a true switching flow if and only if one of the following conditions holds.*
(1) $\mathrm{MRU}(G, F)$ *is acyclic.*
(2) *There is exactly one cycle in* $\mathrm{MRU}(G, F)$*, and the target node x lies on this cycle.*

UEOPL and CLS containment. Gärtner et al. rely on the properties of Lemma 5.6 to prove their containment result [GHH+18]. Having shown the analogue of that lemma for succinct games, we can now follow their proof directly. We summarise the technique here, and direct the reader to Gärtner et al. [GHH+18] for the full details.

Since the game is zero-player, there is a unique play $\pi = (v_1, C_1), (v_2, C_2), \ldots, (v_k, C_k)$ with $v_1 = s$ and $v_k = t$. For each i, the run profile N_i gives is the unique true switching flow that witnesses that the play passes through (v_i, C_i) after i steps. Hence, we can build an (exponentially long) line of switching flows $N_0, N_1, \ldots, N_k$. Given a switching flow N_i, we can compute the next switching flow N_{i+1}, and the previous switching flow N_{i-1}. This is enough to build an End-of-Metered-Line instance, which gives the following result.

Theorem 5.10. *The problem of solving a zero-player succinct switching game is in* `UEOPL` *and* `CLS`*.*

6. Further work

Many interesting open problems remain. For the zero-player case in the *explicit* case, there is an extremely large gap between the upper bounds of `NP` ∩ `coNP` and `PLS` and the easy lower bound of `NL` that we showed here. We conjecture that the problem is in fact `P`-complete, but despite much effort, we were unable to improve upon the upper or lower bounds.

For the one-player case we have shown tight bounds. For the two-player case we have shown a lower bound of `PSPACE` and an upper bounds of `EXPTIME`. We conjecture that the lower bound can be strengthened, since we did not make strong use of the memory requirements that we identified in Section 4.3.

Finally, here we studied the problem of reachability, which is one of the simplest model checking tasks. What is the complexity of model checking more complex specifications?

Acknowledgements

Fearnley was supported by EPSRC grant EP/P020909/1 "Solving Parity Games in Theory and Practice". Mnich was supported by ERC Starting Grant 306465 (BeyondWorstCase) and DFG grant MN 59/4-1. A Visiting Fellowship from the Department of Computer Science at the University of Liverpool allowed Mnich to visit Fearnley, Gairing, and Savani.

References

[AB13] Hoda Akbari and Petra Berenbrink. Parallel rotor walks on finite graphs and applications in discrete load balancing. In *Proc. of SPAA 2013*, pages 186–195, 2013.

[CDFS10] Joshua Cooper, Benjamin Doerr, Tobias Friedrich, and Joel Spencer. Deterministic random walks on regular trees. *Random Structures Algorithms*, 37(3):353–366, 2010.

[CDST07] Joshua Cooper, Benjamin Doerr, Joel Spencer, and Gábor Tardos. Deterministic random walks on the integers. *European J. Combin.*, 28(8):2072–2090, 2007.

[CKS81] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. Assoc. Comput. Mach.*, 28(1):114–133, 1981.

[Con89] Anne Condon. *Computational models of games*. ACM distinguished dissertations. MIT Press, 1989.

[Con92] Anne Condon. The complexity of stochastic games. *Inf. Comput.*, 96(2):203–224, 1992.

[CS06] Joshua Cooper and Joel Spencer. Simulating a random walk with constant error. *Combin. Probab. Comput.*, 15(6):815–822, 2006.

[DF09] Benjamin Doerr and Tobias Friedrich. Deterministic random walks on the two-dimensional grid. *Combin. Probab. Comput.*, 18(1-2):123–144, 2009.

[DGK+16] Jérôme Dohrau, Bernd Gärtner, Manuel Kohler, Jiří Matoušek, and Emo Welzl. ARRIVAL: A zero-player graph game in $\mathtt{NP}\cap\mathtt{coNP}$. Technical report, 2016. https://arxiv.org/abs/1605.03546.

[DGK+17] 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.

[FGHS21] John Fearnley, Paul Goldberg, Alexandros Hollender, and Rahul Savani. The complexity of gradient descent: $\mathtt{CLS} = \mathtt{PPAD} \cap \mathtt{PLS}$. In *Proc. of STOC 2021*, 2021.

[FGMS18] John Fearnley, Martin Gairing, Matthias Mnich, and Rahul Savani. Reachability switching games. In *Proc. of ICALP 2018*, volume 107 of *Leibniz Int. Proc. Informatics*, pages 124:1–124:14, 2018.

[FGMS20] John Fearnley, Spencer Gordon, Ruta Mehta, and Rahul Savani. Unique end of potential line. *J. Comput. Syst. Sci.*, 114:1–35, 2020.

[FGS12] Tobias Friedrich, Martin Gairing, and Thomas Sauerwald. Quasirandom load balancing. *SIAM J. Comput.*, 41(4):747–771, 2012.

[GHH+18] Bernd Gärtner, Thomas Dueholm Hansen, Pavel Hubáček, Karel Král, Hagar Mosaad, and Veronika Slívová. ARRIVAL: Next stop in CLS. In *Proc. of ICALP 2018*, volume 107 of *Leibniz Int. Proc. Informatics*, pages 60:1–60:13, 2018.

[GHH21] Bernd Gärtner, Sebastian Haslebacher, and Hung P. Hoang. A subexponential algorithm for ARRIVAL. Technical report, 2021. https://arxiv.org/abs/2102.06427.

[GHR95] 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.

[GP09] Jan Friso Groote and Bas Ploeger. Switching graphs. *Internat. J. Found. Comput. Sci.*, 20(5):869–886, 2009.

[HLM+08] 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.

[HP10] 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.

[HY17] Pavel Hubáček and Eylon Yogev. Hardness of continuous local search: Query complexity and cryptographic lower bounds. In *Proc. of SODA 2017*, pages 1352–1371, 2017.

[Kar17] Karthik C. S. Did the train reach its destination: The complexity of finding a witness. *Inf. Process. Lett.*, 121:17–21, 2017.

[KNP11] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. of CAV 2011*, volume 6806 of *Lecture Notes Comput. Sci.*, pages 585–591, 2011.

[KRW12] Bastian Katz, Ignaz Rutter, and Gerhard Woeginger. An algorithmic study of switch graphs. *Acta Inform.*, 49(5):295–312, 2012.

[Mei89] Christoph Meinel. Switching graphs and their complexity. In *Proc. of MFCS 1989*, volume 379 of *Lecture Notes Comput. Sci.*, pages 350–359, 1989.

[PDDK96] 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.

[PT87] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of markov decision processes. *Math. Oper. Res.*, 12(3):441–450, 1987.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[Rei09] Klaus Reinhardt. The simple reachability problem in switch graphs. In *Proc. of SOFSEM 2009*, pages 461–472, 2009.

[Tov84] Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Appl. Math.*, 8(1):85–89, 1984.