Connectivity Games over Dynamic Networks

Sten Grüner[a,1], Frank G. Radmacher[b,2,*], Wolfgang Thomas[b]

[a] *RWTH Aachen University, Lehrstuhl für Prozessleittechnik, 52054 Aachen, Germany*
[b] *RWTH Aachen University, Lehrstuhl für Informatik 7, 52056 Aachen, Germany*

Abstract

A game-theoretic model for the study of dynamic networks is proposed and analyzed. The model is motivated by communication networks that are subject to failure of nodes and where the restoration needs resources. The corresponding two-player game is played between "Destructor" (who can delete nodes) and "Constructor" (who can restore or even create nodes under certain conditions). We also include the feature of information flow by allowing Constructor to change labels of adjacent nodes. As an objective for Constructor the network property to be connected is considered, either as a safety condition or as a reachability condition (in the latter case starting from a non-connected network). We show under which conditions the solvability of the corresponding games for Constructor is decidable, and in this case obtain upper and lower complexity bounds, as well as algorithms derived from winning strategies. Due to the asymmetry between the players, safety and reachability objectives are not dual to each other and are treated separately.

Keywords: infinite games, dynamic networks, fault-tolerant systems

1. Introduction

A classical scenario for the application of game-theoretic methods in verification is the antagonism between a possibly malicious *environment* and a *system* (or its control component) that has to guarantee a desired behavior given any choice of actions of the *environment*. The task of verification is then to show that in this game between *environment* and *system* the player *system* has a winning strategy, and in an ideal situation it should even be possible to generate

*Corresponding author
Email addresses: s.gruener@plt.rwth-aachen.de (Sten Grüner), radmacher@automata.rwth-aachen.de (Frank G. Radmacher), thomas@automata.rwth-aachen.de (Wolfgang Thomas)

[1] This author was supported by the DFG Research Training Group "AlgoSyn" (Algorithmic Synthesis of Reactive and Discrete-Continuous Systems), German Research Foundation grant DFG GRK 1298.

[2] This author was supported by the DFG Cluster of Excellence "UMIC" (Ultra High-Speed Mobile Information and Communication), German Research Foundation grant DFG EXC 89.

such a winning strategy from the specification of the desired behavior (defined, e.g., in terms of a formula of temporal logic).

The present article pursues this view in a specific context that is of central interest in the theory of communication networks. We study the antagonism between "suppliers" and "users" of a communication network on one side and the generation of faults (either by nature or by malicious interference) on the other. So, we consider games on dynamic networks, in which a game position is just given by a current shape of a network. The party that generates faults is modeled by a player called *Destructor* who can "delete" nodes in a network. In the present article we only consider changes in the set of nodes (and induced changes in the set of edges). The more general case that arises when including edges in the dynamics involves heavier notation, but does not affect the general results as they appear in this article. (More precisely, our framework is able to simulate edge deletions by modeling each edge as a vertex [1].)

The other parties involved are the suppliers of the network and the users. There are many ways to model these parties. We consider here a model that represents a compromise between conceptual simplicity and adequacy for practical applications.[3] The aspect of simplicity is introduced by a merge of the two parties suppliers and users in a single player called *Constructor*. This player has the power to restore deleted nodes or even to create new nodes (i.e., to extend the network beyond its original shape), matching the purpose of a supplier, and she also can transmit information along edges of the network, matching the actions of an user. The latter feature is realized by the possibility to relabel two nodes that are connected by an edge.

A detailed description of these possible actions by Destructor and Constructor yields a *dynamic network game* in which these players carry out their moves in alternation, step by step changing the shape and the labeling of the network. In the present article we analyze these games only with one objective (winning condition for Constructor), namely with the objective to guarantee that the network is connected. The objective arises in two versions: as a safety game in which connectivity of the network is to be guaranteed forever by Constructor, or as a reachability game in which Constructor has to construct a connected network, starting from a disconnected one. Since we assume complete information, these games are trivially determined. Our aim is to clarify under which assumptions these games are effectively solvable, i.e., that one can decide who wins (and in this case to construct a winning strategy for the winner). One should note that by the independent and very different conceptions of the two players, there is no direct duality between reachability and safety objectives; both games have to be analyzed separately. On the other hand, simplifications are built into our model, for instance by our decision only to declare connectivity of the network as Constructor's objective, i.e., that the aim of the users to realize the transfer of information from certain source nodes to target nodes is not taken into ac-

[3]We thank our colleagues in the research cluster UMIC (Ultra High-Speed Mobile Information and Communication) for their contributions in devising the current model.

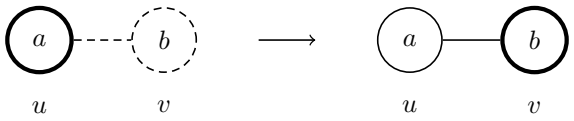

Figure 1: Movement of a strong node u restoring a deleted node v.

count. (For a study of network games including this aspect, see the paper [2].) Another simplification of our model is that Destructor and Constructor have complete information about the network and its current state. Deciding which player wins in a two-player game of incomplete information is known to be much harder [3]. However, providing the Destructor with full information reflects a worst-case behavior as it is often preferred in the domain of formal verification. Regarding the Constructor, one might prefer a model of incomplete information in which the individual users and suppliers are treated as separate players who build a coalition. In general, however, determining the winner in a multiplayer game where the players only have partial information is undecidable [4]. In contrast, in our setting – where players have full information – in many cases we are still able to decide whether Constructor wins.

Before stating our results, let us sketch informally but in a little more detail the definitions of Constructor's actions, which she chooses from a given set of rules. We distinguish three different types of rules. The first is concerned only with the "information flow" through the network evoked by the users; nodes and edges stay fixed. A natural way to describe this aspect is to assume a labeling of the nodes that may change over time. For instance, the label a on node u and a blank label on the adjacent node v are modified to the blank label on u and the label a on v, corresponding to a shift of the data a from u to v. Only the labels of adjacent non-deleted nodes can change; for the same reason as in communication networks only neighboring active clients are able to send or receive messages. A *relabeling rule* describes in which way Constructor may change the labels of adjacent nodes. The other rules entail changes to the network structure. Constructor can restore nodes, which could have existed before, or create completely new nodes. These operations require so-called *strong nodes*; these nodes cannot be deleted and are the prerequisite for restoring and creating nodes. One can view strong nodes as maintenance resources of suppliers, which are located on some places in the network. We often refer to the node property of being strong as the *strongness* property. This property may be moved through edges to existing nodes; being at some node u the strongness can also be used to restore a deleted node v by moving to it if there was an edge (u, v) in the network before v has been deleted (see Figure 1). For the creation of a node we can pick some set S of strong nodes, create a new node v (which may be strong or not) and connect it by an edge with each node in S (see Figure 2). Both actions are either feasible in general or subject to constraints given by the labels of the involved nodes; we will collect these constraints in *movement* and *creation rules*. Since the "information flow"

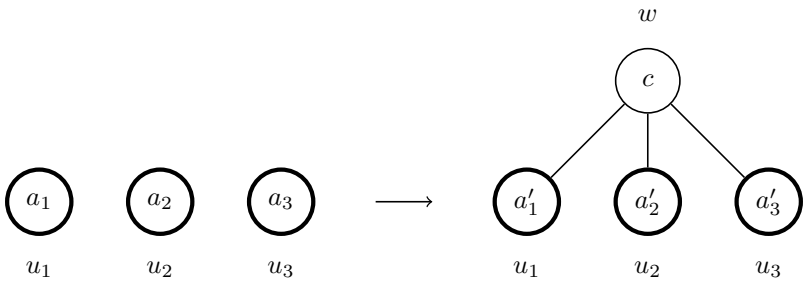

Figure 2: Creation of a new node by a set $U = \{u_1, u_2, u_3\}$ of strong nodes.

in a network should be faster than the maintenance of the network, we allow rules that combine multiple relabeling actions in a single rule, while movement and creation rules only contain single actions. We also take into account that node creation is "more expensive" than restoration. For this reason a creation rule may also change the labels of the involved strong nodes. So, the involved strong nodes may need to be relabeled first before they can be used again for some node creation. We introduce the game model in detail in Section 2.

In this article we are concerned only with the network connectivity. Thus, we consider the connectivity of the network as winning condition, either in a reachability version, where Constructor has to establish a connected network (starting from a disconnected network), or in a safety version, where Constructor has to guarantee that the network is always connected. We analyze the decidability and the computational complexity of solving these safety and reachability games. For the analysis we consider variants of each of these games; there Constructor's rules are restricted to involve only rules of certain types, or moreover, nodes cannot be distinguished by labels. The results differ depending on the considered restriction of the rules, and, perhaps more surprisingly, the results also differ for safety and reachability games. We will show that both solving reachability and solving safety connectivity games are undecidable in general, but for some fragments where Constructor's rules or the labels of nodes are restricted these games become solvable.

In particular, for safety games we show that solvability (for Constructor) is undecidable in the general case but decidable when either creation of new non-strong nodes is disallowed or movement of strong nodes is disallowed. The former problem is PSpace-complete, the latter is solvable in ExpTime (where the input size is given by the size of the initial network and the size of the rule set of Constructor). Maybe surprisingly, solving safety games remains PSpace-complete for those restrictions where node creation is completely forbidden and where, additionally, nodes cannot be distinguished by labels. These results are presented in Section 3.

For solving reachability games, we obtain a stronger undecidability result than for safety games; solving reachability games is also undecidable if Con-

game variant allowed rules	objective	
	reachability	safety
expanding		
w-create, move, relabel	undecidable	undecidable
s-create, w-create, relabel	undecidable	in ExpTime
s-create, move, relabel	undecidable	PSpace-complete
s-create, relabel	undecidable	PSpace-complete
w-create, move	undecidable	open
w-create, relabel	PSpace-hard / in ExpTime	in ExpTime
non-expanding		
move, relabel	PSpace-hard / in ExpTime	PSpace-complete
unlabeled non-expanding	NP-hard / in PSpace	PSpace-complete

Table 1: The complexity of solving reachability and safety connectivity games. We distinguish whether Constructor is allowed to create strong nodes (*s-create*), create weak nodes (*w-create*), *move* strong nodes, or *relabel* nodes.

structor is only allowed to create strong nodes and relabel nodes or she is only allowed to create non-strong nodes and move strong nodes. Moreover, these results still hold for solitaire games in which Destructor always skips. For the restrictions where creation of nodes is completely disallowed or where both the creation of strong nodes and relabelings are disallowed, we obtain partial results by providing lower and upper bounds; in these two cases solvability by Constructor is PSpace-hard (while the easily obtained upper bound is ExpTime). For the case where not only node creation but also node labels are suppressed, we obtain that solving reachability games is in PSpace and NP-hard. The proofs illustrate again the difference between safety and reachability. We present these results in Section 4.

An overview of the results is given in Table 1. In Section 5 we also discuss some cases in which solutions remain open. We also mention some selected perspectives from a very rich landscape of problems that remain to be treated in this area combining practical issues with theoretical research.

Related Work. The game-theoretic approach is inspired by *sabotage games*, which van Benthem suggested in [5]. There, a reachability problem over graphs is considered, where a "Runner" traverses a graph while a "Blocker" deletes an edge after each move. The theory of these games and many variants have been thoroughly studied in [6, 7, 8, 9, 10, 11].

To the best of our knowledge, other contributions are only loosely related to our game-theoretic analysis of dynamic networks. For the sake of completeness, we mention in the following some other approaches to analyze dynamically changing systems and, especially, dynamic networks.

Most of the literature about games on networks deals with games in strategic form and the computation of Nash equilibria that correspond to stable points of

network operation (see [12]); these approaches do not consider the evolvement of a network in which agents have to react on faults.

Dynamically changing systems are also addressed by *online algorithms* (see [13, 14, 15]). These find applications in routing and scheduling problems in wireless and dynamically changing wired networks (see [16, 17]). However, the only approaches we are aware of where a adversary also changes the network structure is due to Awerbuch et al. [18, 19]; there a routing objective is faced with an adversary that injects packets and also decides which connections are available. These studies aim at a competitive analysis of the "communication throughput"; the number of delivered packets of an online algorithm is compared to an optimal offline algorithm.

Another view on online algorithms are *dynamic algorithms* (see [20, 21]). A *fully dynamic algorithm* refers to a dynamic graph in which edges are inserted and deleted; the focus of investigation is the computational complexity of static graph properties with respect to a given sequence of update steps (see [22, 23]). The same idea leads to a *dynamic complexity theory*, which deals with the complexity of computing and maintaining an auxiliary structure; this structure allows to extract the solution of a decision problem for a dynamically changing instance (see [24]).

Studies on a game-theoretic model for routing under adversarial condition have been started in [2] (also see [25]). Instead of a competitive analysis of a given online algorithm, the aim is to check whether a given dynamic scenario has a solution in form of a routing scheme (and to synthesize a routing scheme if it exists). This model is also inspired by the sabotage game model, but complementary to the present work. The adversary deactivates edges and injects packets in the network, and a solution of the game requires that all packets must be delivered or that the overall number of packets in the network is bounded.

Another interesting approach arises from the studies of dynamic versions of the *dynamic logic of permission* (DLP), which is in turn an extension of the *propositional dynamic logic* (PDL). In DLP, "computations" in a Kripke structure from one state to another are considered which are subject to "permissions" [26]. The logic $\mathrm{DLP}^{+}_{\mathrm{dyn}}$ (see [27, 28]) extends DLP with formulas which allow updates of the permission set and thus can be seen as a dynamically changing Kripke structure. However, the dynamic changes have to be specified in the formula; an adversarial agent is not considered.

The idea of changing networks is of course studied in considerable depth in the theory of graph grammars, graph rewriting, and graph transformations (see [29, 30, 31]). While there the class of generable graphs (networks) is the focus of study, we deal here with the more refined view when considering the evolvement of a two-player game and the properties of graphs occurring in them. In the (one-player) framework of model checking, we mention the work [32], where *graph-interpreted temporal logic* is introduced as a rule-based specification. They developed a technique to map a "graph transition system" (where each node is a graph) to a finite Kripke structure, so that classical LTL model checking can be applied. In contrast, an approach to model checking of dynamically changing Kripke structures is *module checking* [33]. There, in the context of open (reac-

tive) systems, the environment can remove (all but one) and restore successor nodes in a Kripke structure. However, the dynamic changes are enforced by a single player, the environment.

Bibliographic Note. Most of the results in this article has been published in the papers [34] and [35]. We extend our prior results and provide a comprehensive and polished presentation.

2. The Connectivity Game Model

A *dynamic network connectivity game* (or short *connectivity game*) is played by two players, *Destructor* and *Constructor*, who modify a special kind of graph, called *network*, starting from an *initial network* G. While in every turn Destructor can delete one of the nodes in the network, which does not correspond to a maintenance resource, Constructor's moves are subject to a given set of *rules* R. Formally, a dynamic network connectivity game is a pair

$$\mathcal{G} = (G, R)$$

consisting of an initial network G and a finite set R of rules for Constructor. In the following we will define networks, the possible moves of the two players, and the rules that may be contained in Constructor's rule set.

Networks. To define networks we require an enriched graph structure that captures the features of node labels, deactivated (deleted) nodes, and maintenance resources (strong nodes). Formally, we present *networks* in the form

$$G = (V, E, A, S, (P_a)_{a\in\Sigma})$$

with

- a finite set V of vertices (also called nodes),
- an undirected edge relation $E \subseteq V \times V$,
- a set $A \subseteq V$ of *active nodes*,
- a set $S \subseteq A$ of *strong nodes*,
- a partition of V into sets P_{a_i} for some label alphabet $\Sigma = \{a_1, \ldots, a_k\}$. A node that belongs to P_a carries the label a.

We say that a node is *deactivated* or *deleted* if it is not active. A *weak node* is an active node which is not strong. A network is connected if the graph that is induced by the active vertices is connected, i.e., for any two active vertices u, v there exists a path from u to v which only consists of active nodes.

Moves and Rules. In a dynamic network connectivity game $\mathcal{G} = (G, R)$ the dynamics arises from the initial network G by the moves of *Destructor* and *Constructor*, which make their moves in alternation; Destructor starts. Also, both players are allowed to skip at each turn. A *play* of a game $\mathcal{G}$ is an infinite sequence

$$\pi = G_1 G_2 G_3 \cdots$$

where G_1 is the initial network and each step from G_i to G_{i+1} results from the move of Destructor (if i is odd) or Constructor (if i is even). If a player skips in turn i, the network does not change, i.e., $G_i = G_{i+1}$. So, plays are infinite in general, but may be considered finite when neither of the players can move anymore or a given objective (winning condition) is satisfied.

In the following we describe the players' moves in detail. When it is Destructor's move, he can perform a *deletion step* by deleting some weak node $v \in A \setminus S$; the set A is changed to $A \setminus \{v\}$. When it is Constructor's move, she can choose a rule from her rule set R that is applicable on the current network; then she selects vertices that match this rule. The rules in R can be of three different types, which are described in the following.

Relabeling rule: A rule $\langle\, a, b \longrightarrow c, d \,\rangle$ allows Constructor to change the labels a and b of two active adjacent nodes in A into c and d, respectively. Formally, for two vertices $u \in P_a$ and $v \in P_b$ with $(u, v) \in E$ the sets P_a, P_b, P_c, and P_d are updated to $P_a \setminus \{u\}$, $P_b \setminus \{v\}$, $P_c \cup \{u\}$, and $P_d \cup \{v\}$.

For relabeling rules we will also consider rules with multiple relabelings in one turn. This corresponds to our intuition that there can be a lot of information flow in the network at the same time. For example, for two relabeling steps in one turn we use the notation

$$\langle\, a, b \longrightarrow c, d \,;\; e, f \longrightarrow g, h \,\rangle \ .$$

Constructor applies the relabelings one after the other, but in the same move.

Movement rule: A rule $\langle\, a \xrightarrow{\text{move}} b \,\rangle$ allows Constructor to *shift the strongness* from a strong node that carries the label a to an adjacent node that is labeled with b and must not be strong. Formally, for two vertices $u \in P_a$ and $v \in P_b$ with $u \in S$, $v \notin S$, and $(u, v) \in E$, the set S is updated to $(S \setminus \{u\}) \cup \{v\}$ and A is updated to $A \cup \{v\}$. The case $v \in A$ means to simply shift strongness to v; the case $v \in V \setminus A$ means *restoration* of v, which is illustrated in Figure 1. Here we use both terms "moving a strong node" and "shifting its strongness" with exactly the same meaning.

Creation rule: These rules enable Constructor to create a completely new node, which was not in V before. A rule

$$\langle\, a_1, \ldots, a_n \xrightarrow{\text{create}(c)} a'_1, \ldots, a'_n \,\rangle$$

allows Constructor to choose any set $U = \{u_1, \ldots, u_n\} \subseteq S$ of n different strong nodes such that the label of u_i is a_i (for all $i \in \{1, \ldots, n\}$). Then,

Constructor creates a new active node w, labels it with c, and connects it to every node in U. Formally, the sets V and A are updated to $V \cup \{w\}$ and $A \cup \{w\}$, respectively; also E is updated by adding edges between w and each node of U. Also the labels of the nodes in U may change after creation; the label of u_i is changed to a'_i (for all $i \in \{1, \ldots, n\}$). For $n = 3$ this is depicted in Figure 2.

For the *creation of a strong node* we use the notation

$$\langle\, a_1, \ldots, a_n \xrightarrow{\text{s-create}(c)} a'_1, \ldots, a'_n \,\rangle \ .$$

In this case also S is updated to $S \cup \{w\}$.

Note that a creation rule may also change the labels of the strong nodes in U. So, possibly these nodes have to be relabeled first before they can be used again for the same node creation. This corresponds to our intuition that node creation causes higher costs than restoration.

We consider some variants where Constructor's moves are restricted. A game (G, R) is called *non-expanding* if R does not contain any creation rule. In *unlabeled non-expanding* games, nodes can never be distinguished by their labels; formally, we assume that all vertices are labeled with a blank symbol $\sqcup$ and the movement rule $\langle \sqcup \xrightarrow{\text{move}} \sqcup \rangle$ is the only available rule.

Winning Conditions. For dynamic network connectivity games we only analyze the connectivity of the network (more precisely, of the active nodes). We consider this connectivity property either as a reachability objective or as a safety objective for Constructor. So, we can consider a dynamic network connectivity game $\mathcal{G} = (G, R)$ either as a *reachability connectivity game* or as a *safety connectivity game*. In the former the initial network is disconnected, and Constructor's objective is to reach a connected network. We say that *Constructor wins a play π of the reachability connectivity game $\mathcal{G}$* if π contains a connected network; Destructor wins otherwise. Conversely, in the safety game the initial network is connected, and Constructor has to guarantee that the network always stays connected. So, *Constructor wins a play π of the safety connectivity game $\mathcal{G}$* if all networks in π are connected; otherwise Destructor wins.

Strategies. A *strategy for Destructor* is a function (here denoted by σ) that maps each play prefix $G_1 G_2 \cdots G_i$ with an odd i to a network G_{i+1} that arises from G_i by a node deletion. Analogously, *strategy for Constructor* is a function τ that maps each play prefix $G_1 G_2 \cdots G_i$ with an even i to a network G_{i+1} that arises from G_i by applying one of the rules from R. A strategy is called *positional* (or *memoryless*) if it only depends on the current network, i.e., it is a function that maps the current network G_i to G_{i+1} as above. We say that *Destructor wins the reachability (safety) game* if he has a strategy σ to win every play of the reachability (safety) game in which he moves according to σ. Analogously, *Constructor wins the reachability (safety) game* if she has a strategy τ to win every play of the reachability (safety) game. We call a strategy with which a particular player wins a *winning strategy* for this player.

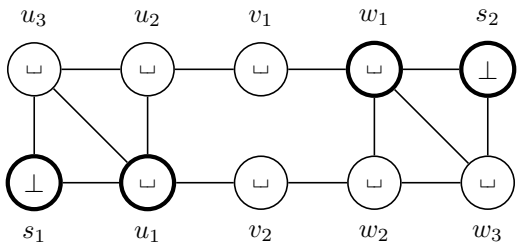

Figure 3: The initial network of a safety connectivity game.

In this article we study connectivity games only with reachability and safety objectives. For these winning conditions it is well known that one can restrict both players' winning strategies to positional strategies (see [36, 37, 38]), i.e., if Constructor (Destructor) wins a game $\mathcal{G}$, she (he) also has a positional strategy to win $\mathcal{G}$. Therefore, we will always assume positional strategies in this article.

The Problem of Solving Connectivity Games. This article mainly deals with the problem of solving a given connectivity game. More precisely, we analyze the following decision problems.

- *Solving reachability connectivity games:* Given a connectivity game $\mathcal{G}$, does Constructor win the reachability connectivity game (i.e., does Constructor have a strategy to eventually reach a connected network)?
- *Solving safety connectivity games:* Given a connectivity game $\mathcal{G}$, does Constructor win the safety connectivity game (i.e., does Constructor have a strategy to guarantee that the network always stays connected)?

Usually, a solution of a game $\mathcal{G}$ comprises both the winner of $\mathcal{G}$ and a winning strategy for the player who wins. To classify this problem in terms of computational complexity [39], we only formulate the question of whether Constructor wins $\mathcal{G}$ as a decision problem. Nevertheless, the solutions for connectivity games that we shall present in Sections 3 and 4 can be adapted to compute a winning strategy.

Example 2.1. As a first example we consider a safety connectivity game, where Constructor has to guarantee that the network always stays connected. The game is played on the network $G = (V, E, A, S, (P_a)_{a\in\Sigma})$ which is depicted in Figure 3. The nodes are labeled over the alphabet $\Sigma = \{\bot, \sqcup\}$. The nodes in $S = \{s_1, s_2, u_1, w_1\}$ are strong; all other nodes are weak. As a scenario for this game one could imagine two clients s_1, s_2 communicating over a network via unreliable intermediate nodes; though the clients are supported by two mobile maintenance resources (initially located on u_1 and w_1). Formally, we define the dynamic network connectivity game $\mathcal{G} = (G, R)$, where the initial network is the depicted network G. We will see that Constructor will only be able to maintain this network depending on her rule set R.

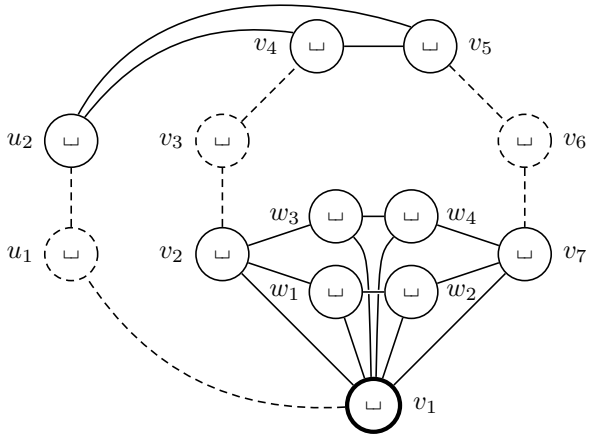

Figure 4: The initial network of an unlabeled reachability game.

First, we consider the rule set R that consists of the rule $\langle \sqcup \xrightarrow{\text{move}} \sqcup \rangle$ only. It means that the strong nodes s_1 and s_2 are not able to move because their labels do not match the movement rule. By taking a closer look at this example we see that Destructor has a winning strategy. He deletes w_3 in his first move; then, we distinguish between two cases. If Constructor restores w_3, Destructor deletes v_1 in his next move and finally u_1, v_2 or w_2. If Constructor does not move the upper movable strong node to w_3, the node w_1 has to remain strong; otherwise Constructor loses by deletion of w_1. It is easy to see that in this case Destructor wins by suitable deletions of nodes in $\{u_1, u_2, v_1, v_2\}$.

As a variant of this example, let us consider the same safety game, but with an additional creation rule: $\langle \sqcup, \sqcup \xrightarrow{\text{create}(\sqcup)} \sqcup, \sqcup \rangle$. We claim that now Constructor has a winning strategy. Whenever Destructor deletes a node, Constructor uses the creation rule to create a new vertex, say v_3, which establishes a new connection between the two strong nodes u_1 and w_1. Even if the newly created node is deleted, Constructor creates a new node again and again. Note that in this way the number of vertices in the set V can increase to an unbounded number.

Example 2.2. As an example for a reachability game, where Constructor has to establish a connected network, we consider a game $\mathcal{G} = (G, R)$ on the unlabeled network G which is depicted in Figure 4. The nodes u_1, v_3, v_6 are deactivated, and the node v_1 is strong. Since this is an unlabeled non-expanding game, all nodes of G carry the same label $\sqcup$, and the rule set R consists of the single rule $\langle \sqcup \xrightarrow{\text{move}} \sqcup \rangle$.

We claim that Constructor wins this reachability game. First of all, Destructor, who tries to keep the network disconnected, has to delete the node u_2; otherwise Constructor could establish a connected network by shifting the only strong node from v_1 to u_1. If Destructor deletes u_2, Constructor moves the strong node from v_1 to v_2. Then, Destructor has to delete v_4 in the following

turn; otherwise Constructor restores the node v_3 and hence wins. So, we can assume that after Destructor's first two deletion steps exactly the vertices in the set $\{u_1, u_2, v_3, v_4, v_6\}$ are deactivated. Then, Constructor moves the strong node from v_2 via v_1 to v_7 in her next two moves. We note that after these moves the subgraph induced on G by the vertex subset $\{v_1, v_2, v_7, w_1, w_2, w_3, w_4\}$ is still connected, also when Destructor deletes two arbitrary nodes. Also, Destructor will not delete v_5 because this node deletion leads immediately to a connected network (since u_2 and v_4 are already deactivated). But then Constructor wins by moving the strong node from v_7 to v_6.

This example gives rise to two notable remarks. First, it is mentionable that in a reachability game it may be worse for Destructor to delete a node than to skip. The reason for this is that a node deletion may decrease the number of connected components in the subgraph induced by the active vertices. The second remark is that it may be necessary for Constructor to shift a strong node to a certain vertex more than once, i.e., to shift a strong node in a loop. Moreover, it may happen that she has to shift a strongness in a loop even if she does not restore any deactivated node with these moves. In the example Constructor moves the strong node from v_1 to v_2 and then back to v_1 without restoring any vertex. Constructor does not have a winning strategy which also guarantees that the strongness visits each vertex at most once.

3. Solvability of Safety Connectivity Games

In this section we analyze the problem of solving safety connectivity games, for which we show in our first result that it is undecidable in general (Section 3.1). Later we point out decidable subcases. We show upper bounds on the complexity of solving these safety games (Section 3.2) and a tight lower bound that even holds in the unlabeled non-expanding case (Section 3.3).

3.1. The general case

To show that solving safety connectivity games is undecidable we construct a safety game to simulate a Turing machine. In this game Constructor is able to keep the network always connected exactly if the Turing machine never halts. It is indeed remarkable that we need weak creation, movement, and relabeling rules for this construction. Later we will see that solving safety games becomes decidable if weak creation or movement rules are absent.

Theorem 3.1. *Solving safety connectivity games is undecidable, even if Constructor can only apply weak creation, movement, and relabeling rules.*

Proof. We reduce the halting problem for Turing machines to the problem of solving safety connectivity games. Here, we present Turing machines in the format

$$M = (Q, \Gamma, \delta, q_0, q_{\text{stop}})$$

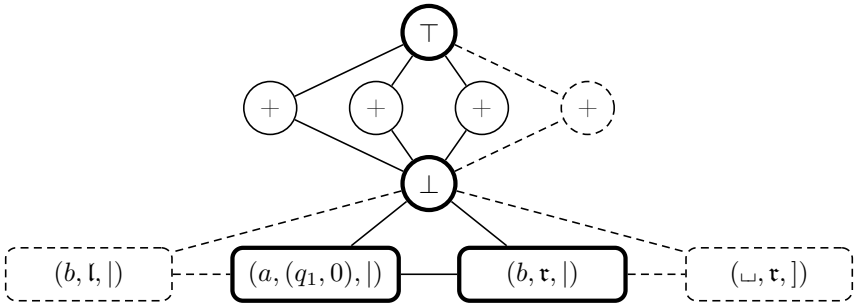

Figure 5: A network of a safety game representing a configuration of a Turing machine in state q_1 with its head over a on a tape containing $bab␣$.

with a state set Q, a tape alphabet Γ (including a blank symbol $␣$), a transition function $\delta\colon Q \setminus \{q_{\text{stop}}\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$, an initial state q_0, and a stop state q_{stop}.

For a Turing machine M we construct a game $\mathcal{G} = (G, R)$ such that M halts when started on the empty tape iff Constructor is not able to keep the network always connected by applying the rules of R, i.e., Destructor wins the safety game $\mathcal{G}$. The idea is to consider a configuration of M as a connected network where Constructor creates additional vertices during the simulation of a valid computation of M. If M stops, she cannot create vertices anymore, and Destructor is able to disconnect the network. We label the nodes that correspond to a configuration of M with triples of the from $\Gamma \times (\widehat{Q} \cup \{⟨, ⟩\}) \times \{\,|\,, \,]\,\}$ with $\widehat{Q} \coloneqq Q \times \{0, 1, \triangleleft, \triangleright\}$. The first component of each node label holds the content of its represented cell of the tape. The second component is labeled with $⟨$ if the represented cell is on the left-hand side of the head and with $⟩$ if the represented cell is on the right-hand side of the head (the information given by the labeling of the nodes with $⟨$ or $⟩$ and by the edges between nodes is sufficient to recover the total order on the cells of the tape); the second component is labeled with $q \in Q$ and some auxiliary symbol if M is in state q and the head is on the cell represented by this node. The third element is either an end marker ( ]) or an inner marker (|) depending on whether the node is the currently the right-most represented cell of the tape or not. Since each of these nodes represents a cell of the tape, we will refer to these nodes as *cell nodes*. Additionally, the label alphabet contains the symbols $\top$, $\bot$, $+$, and !. The labels $\top$, $\bot$ are used for the two additional strong nodes that Constructor has to keep connected; the $\bot$-labeled node is always connected to every cell node while the $\top$-labeled node is only connected to the $\bot$-labeled node via some weak nodes that are labeled with $+$. The exclamation mark (!) is used as a label that Destructor has to prevent to occur; if Constructor manages to relabel a strong node to a !-labeled node, she has a winning strategy regardless of the behavior of M. An example of a network that represents a configuration of a Turing machine is given in Figure 5; there, the tape contains the sequence $bab␣$, the Turing machine is in state q_1, and its head is on the cell containing the a.

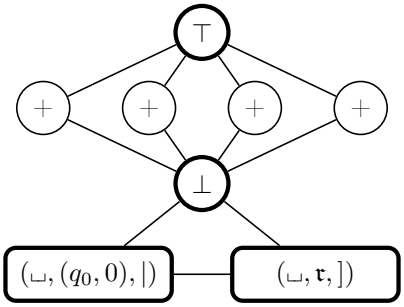

Figure 6: The initial network of a safety game representing the initial configuration of a Turing machine M.

Constructor has to create a $+$-labeled weak node in every turn where she simulates a transition of M. Since we want Constructor to simulate valid transitions only, we ensure that an accordant creation rule can only be applied to the cell node carrying the current state of M and to an adjacent cell node. For this reason only two cell nodes are strong at any time. Constructor is able to shift these strong nodes depending on whether she wants to simulate a left or a right transition of M. We ensure that Constructor shifts the nodes at most once between simulating two transitions; otherwise she would be able to shift them forever instead of simulating M. For this reason the cell node representing the head has auxiliary symbols in $\{0, 1, \triangleleft, \triangleright\}$. The symbol 0 means that Constructor can choose either to move the strong nodes or to simulate a transition. If this symbol is 1, she has already shifted the strong nodes and now must simulate a transition. The symbols $\triangleleft$ and $\triangleright$ are used as intermediate labels when Constructor moves the strong nodes to the left and to the right, respectively. The initial network, which corresponds to the initial configuration of M on an empty working tape, is depicted in Figure 6.

In the following we describe the rule set R. As mentioned before, the rule

$$\langle\, !, \top, \bot \xrightarrow{\text{create}(+)} !, \top, \bot \,\rangle$$

allows Constructor to ensure the connectivity of the network if a strong node obtains the !-label. To allow Constructor to shift the two strong cell nodes to the right, we add the following rules for all $q \in Q$, $a, b \in \Gamma$, and $* \in \{\, |\,, \,]\,\}$:

1. $\langle\, (a, (q, 0), |), \top, \bot \xrightarrow{\text{create}(+)} (a, (q, \triangleright), |), \top, \bot \,\rangle$,
2. $\langle\, (a, (q, \triangleright), |) \xrightarrow{\text{move}} (b, \mathfrak{r}, *) \,\rangle$,
3. $\langle\, (a, \mathfrak{l}, |) \xrightarrow{\text{move}} (b, (q, \triangleright), |) \,\rangle$, and
4. $\langle\, (a, (q, \triangleright), |), \top, \bot \xrightarrow{\text{create}(+)} (a, (q, 1), |), \top, \bot \,\rangle$.

The rules for shifting the two strong nodes to the left are built analogously. (Note that these sequences of rules are only used to *prepare* the simulation of a transition; as explained beforehand, they can be applied between the simulation of two transitions to make the cell node strong to which the head moves next.)

Whenever Constructor applies the second or the third rule, we want to force Destructor to deactivate the weak cell node (instead of a +-labeled node). For this reason we add the relabeling rule

$$\langle\, (a, \mathfrak{l}, |), (b, (q, z), |) \longrightarrow !, ! \,;\; !, (c, \mathfrak{r}, *) \longrightarrow !, ! \,\rangle$$

for every $a, b, c \in \Gamma$, $z \in \{\triangleleft, \triangleright\}$, and $* \in \{\,|\,, \,]\,\}$. Constructor can apply this rule iff a series of three cell nodes is active; it leads to an !-labeled strong node and hence to a network where Constructor wins.

A transition of M is simulated by changing the labels of the two strong cell nodes. One of the cell nodes has to carry, besides the state of M, the auxiliary symbol 0 or 1; in this case it is guaranteed that the two strong cell nodes are adjacent. Due to the rules for moving these strong nodes we can assume that these strong nodes are already at their desired position. Then, it is easy to supply a set of creation rules that mimics the transitions of M. Formally, for each tuple (q, a, p, b, X) with $\delta(q, a) = (p, b, X)$ and for every $c \in \Gamma$, $z \in \{0, 1\}$, and $* \in \{\,|\,, \,]\,\}$ we add the rule

$$\langle\, (c, \mathfrak{l}, |), (a, (q, z), *), \top, \bot \xrightarrow{\text{create}(+)} (c, (p, 0), |), (b, \mathfrak{r}, *), \top, \bot \,\rangle$$

if $X = L$, and

$$\langle\, (a, (q, z), |), (c, \mathfrak{r}, *), \top, \bot \xrightarrow{\text{create}(+)} (b, \mathfrak{l}, |), (c, (p, 0), *), \top, \bot \,\rangle$$

if $X = R$.

Finally, rules are needed to extend the network in the case that more space on the tape is needed. New cell nodes are allocated next to the end marker, which represents the rightmost used cell of the tape. For this allocation we add the rule

$$\langle\, \bot, (a, (q, 0),]) \xrightarrow{\text{create}((\sqcup, \mathfrak{r},]))} \bot, (a, (q, 0), |) \,\rangle$$

for every $a \in \Gamma$, and $q \in Q$. Destructor will deactivate the created node with the label $(\sqcup, \mathfrak{r},])$ immediately to prevent Constructor from relabeling a strong cell node to a !-labeled node.

To show the correctness of the construction, we first assume that M never stops. Constructor continuously simulates the computation of M in order to create new +-labeled nodes, each of which connects the $\top$- with the $\bot$-labeled node; otherwise Destructor wins by deleting all of these nodes. We argue that Constructor can guarantee that there is at least one active +-labeled node, which connects the nodes labeled $\top$ and $\bot$. In the first turn Destructor deletes one of the three +-labeled nodes that are active in the initial network. Destructor may delete another of these nodes if he misbehaves after some tape extension or a strong node shift, but in this case Constructor obtains a strong !-labeled node. So, Destructor can only reduce the number of +-labeled nodes to one in the following move, before Constructor can produce a new node connecting the $\top$- with the $\bot$-labeled node in every turn from this point onwards. Thus, whenever Constructor shifts a strong cell node, Destructor has to deactivate the node

where this strongness was shifted from; whenever Constructor creates a new cell node, Destructor has to deactivate this node immediately; and whenever Constructor simulates a transition, she creates a new +-labeled node. Since M never stops, Constructor keeps the network connected by simulating M.

Conversely, if M stops, Constructor cannot apply any rule for simulating a transition from some point onwards. The construction ensures that Constructor can shift the strong cell nodes or create a new cell node at most once after simulating a transition. So, Constructor can only skip from some point onwards. Hence, Destructor wins by deleting all +-labeled nodes. □

3.2. Decidable Subcases

Now, we analyze safety games under some restrictions of the given rule set. If we prohibit weak creation rules, solving safety games is PSPACE-complete (where the input size is given by the size of the initial network and the size of the rule set of Constructor). The PSPACE-hardness also holds in the more restricted unlabeled non-expanding case (see Theorem 3.7). In the following we show the inclusion in PSPACE. The basic idea is to show that the we can already determine the winner after a number of turns that is polynomial in the size of the given game. To follow this idea we observe that it is optimal for Destructor to delete a node in every turn. As a consequence the number of weak nodes can be assumed to be monotonically decreasing. This allows us to bound the number of turns that Destructor needs to win a game (if he has a winning strategy) to a number that is polynomial in the size of the given game. This bound allows us to solve safety connectivity games by traversing the game tree in a depth-first manner.

We call a strategy of Destructor *strict* if he deletes a vertex in every turn (i.e., he does not skip) whenever there is still a weak node left for deletion. We can assume that Destructor always plays a strict strategy in a safety game: if Destructor skips, so Constructor can skip as well leading the play to the same network (which is still connected).

Remark 3.2. If Destructor wins a safety connectivity game $\mathcal{G}$, he also has a strict strategy to win $\mathcal{G}$.

For a play $\pi = G_1 G_2 \dots$ we define the *level* of a network G_i as the number of weak nodes in G_i if Destructor acts next (i.e., i is odd) and as the number of weak nodes in G_i plus 1 if Constructor moves next (i.e., i is even). Clearly, if Destructor plays according to a strict strategy, the level is monotonically decreasing as long as the level has not reached 0 (or Destructor has won).

Lemma 3.3. *Consider a safety connectivity game $\mathcal{G}$ without weak creation rules. If Destructor wins $\mathcal{G}$, he also wins $\mathcal{G}$ with a strict strategy such that, for each ℓ, Constructor is able to shift each strongness at most $n_\ell \cdot d_\ell$ times in networks of level ℓ before a disconnected network is reached, where n_ℓ (d_ℓ) is the number of nodes (deactivated nodes) of the first occurring network of level ℓ.*

Proof. Assume that Destructor wins, say with a strict winning strategy σ (which he has due to Remark 3.2). Towards a contradiction, also assume that Constructor has a strategy τ where, for some ℓ, she is able to shift a strongness more than

$n_\ell \cdot d_\ell$ times in networks of level ℓ before Destructor wins. Now, consider a play π where Destructor and Constructor play according to σ and τ, respectively. So, there exists some ℓ such that Constructor shifts a strongness at least $n_\ell \cdot d_\ell + 1$ times in networks of level ℓ. Let G_i be the first network of level ℓ in π, and let G_k be the last network of level ℓ in π, where either Destructor has already won (i.e., G_k is disconnected) or Constructor's move decreases the level. We note that applying a strong creation rule would immediately decrease the level to $\ell - 1$; and weak creation rules, which preserve the level, are forbidden. So, since Destructor's strategy σ is strict, we know that Constructor only applies movement rules in the play infix $G_i \cdots G_k$. Hence, the set of nodes and their labels are preserved in this play infix.

In the play infix $G_i \cdots G_k$ each strongness is shifted along a certain path of nodes, each of which must have been deactivated before Constructor shifts the strongness to it; otherwise the level would decrease to $\ell - 1$ immediately. Among these deactivated vertices we distinguish, for each network in $G_i \cdots G_k$, between the nodes that have already been deactivated since G_i and the nodes that have been deleted by Destructor in some network of level ℓ at least once. As the network G_i consists of d_ℓ deactivated nodes, in the play infix $G_i \cdots G_k$ Constructor shifts a strongness at most d_ℓ times to a node that has not been deleted by Destructor in some network of level ℓ before. Since there is a strongness that Constructor shifts at least $n_\ell \cdot d_\ell + 1$ times in networks of level ℓ, there is a play infix $G_{j_1} \cdots G_{j_2}$ of π with $i \leq j_1 < j_2 \leq k$ where a strongness is shifted in a loop such that the node where this strongness is shifted to has been deleted by Destructor before in some network of level ℓ. Assume that this loop consists of m nodes. Since Constructor restores these m nodes, none of these m nodes stays deactivated until Destructor wins or the level decreases.

It remains to be shown that Destructor does not have to delete all of these m nodes in order to prevent Constructor from applying a certain rule. By definition the m deleted nodes are restored by the same strongness; none of the other strong nodes has to be moved in order to restore them. The vertices, edges, and labels of the network stay unchanged during the loop. So, Constructor's possibilities for node creation and movement are not constricted. It remains the case that Destructor has to delete all of the m nodes to prevent Constructor from applying a relabeling rule. In this case we obtain a winning strategy for Constructor since she would be able to move the strong node in the loop again and again, which would take her as many turns as Destructor needs for the node deletions (in this case Destructor would not be able to perform any other node deletion).

Therefore, at least one of these m node deletions is needless for Destructor; we can eliminate it from Destructor's strategy without harming his strict winning strategy. (For the elimination step, we let Destructor successively delete the next weak node that he would delete by playing his strategy σ.) We can optimize Destructor's strategy by repeating this elimination step. This improvement process is finite, because the resulting play changes only from the point onwards where we change Destructor's strategy (and Destructor eventually reaches a disconnected network). Destructor still wins with the resulting

strategy and additionally prevents for all ℓ that any strongness is shifted more than $n_\ell \cdot d_\ell$ times in networks of level ℓ. This is a contradiction to our assumption. □

So, for safety games where Destructor wins, we obtained an upper bound to the length of any path along which a certain strongness can be shifted within the same level. From this we can derive an upper bound for the number of node deletions that Destructor needs to win.

Lemma 3.4. *Consider a safety connectivity game $\mathcal{G}$ without weak creation rules. Let $|V|$ ($|S|$) be the number of active nodes (strong nodes) of the initial network. If Destructor wins $\mathcal{G}$, he also has a strict strategy to win $\mathcal{G}$ with at most $|S| \cdot (2|V| - |S|)^3$ node deletions.*

Proof. Assume that Destructor wins the safety game $\mathcal{G}$. The previous lemma states that Destructor also wins with a strict strategy where, for each ℓ, Constructor can shift each strongness at most $n_\ell \cdot d_\ell$ times in networks of level ℓ. Since the number of strong nodes is fixed, Destructor wins with a strict strategy where, for each ℓ, he acts at most $|S| \cdot n_\ell \cdot d_\ell = |S| \cdot n_\ell \cdot (n_\ell - |S| - \ell)$ times in networks of level ℓ. For strict strategies the level is monotonically decreasing (as long as it has not reached 0). The level decreases at most $|V| - |S|$ times; so, for every ℓ we can overapproximate the total number of nodes in a network of level ℓ by $n_\ell \leq |V| + (|V| - |S|) = 2|V| - |S|$. Hence, Destructor wins with a strict strategy deleting at most

$$\begin{aligned}
&\sum_{\ell=0}^{|V|-|S|} |S| \cdot n_\ell \cdot (n_\ell - |S| - \ell) \leq \sum_{\ell=0}^{|V|-|S|} |S| \cdot n_\ell \cdot (n_\ell - |S|) \\
&\leq (|V| - |S| + 1) \cdot \big(|S| \cdot (2|V| - |S|) \cdot (2|V| - |S| - |S|)\big) \\
&\leq |S| \cdot (2|V| - |S|)^3
\end{aligned}$$

nodes. □

To show that we can solve safety connectivity games is in PSpace (if weak creation rules are forbidden) it suffices to build up the game tree, which we truncate after $|S| \cdot (2|V| - |S|)^3$ moves of Destructor. We construct the game tree on-the-fly in a depth-first manner; so, we only have to store a path from the root to the current node, which length is polynomial in the size of $\mathcal{G}$.

Theorem 3.5. *In the case that Constructor does not have any weak creation rule, solving safety connectivity games is in* PSpace.

Proof. Consider a safety connectivity game $\mathcal{G}$ without weak creation rules. We build up the game tree $t_\mathcal{G}$; each node of $t_\mathcal{G}$ consists of a network and the move number. The root of $t_\mathcal{G}$ is the initial network with move number 1. The successors of a node with an odd move number i arise by all possible moves of Destructor and have the even move number $i+1$. Analogously, the successors of a node with an even move number i arise by all possible moves of Constructor and have

the odd move number $i+1$. We build $t_{\mathcal{G}}$ up to height $k := 2 \cdot |S| \cdot (2|V| - |S|)^3$. So, each node of $t_{\mathcal{G}}$ with move number k is a leaf. Also, each node of $t_{\mathcal{G}}$ that consists of a disconnected network is a leaf. Since $t_{\mathcal{G}}$ is finitely branching, $t_{\mathcal{G}}$ is finite.

We can construct $t_{\mathcal{G}}$ on-the-fly in a depth-first manner; at each backtracking step we label the current node v with 1 if Constructor wins form v and with 0 otherwise. If v is a leaf, we label it with 1 iff v consists of a connected network. We label an inner node v that has an odd move number with 1 iff all successors are labeled with 1. Analogously, we label an inner node v that has an even move number with 1 iff it has at least one successor labeled with 1. Using the bound from the previous lemma it follows that Constructor wins the safety game on $\mathcal{G}$ iff the root of $t_{\mathcal{G}}$ is labeled with 1.

In order to compute $t_{\mathcal{G}}$ as described, we only have to store a path from the root to the current node and some auxiliary information about the labels of the successors of the current node. Since the height of $t_{\mathcal{G}}$ is at most k, the required space is polynomial in the size of $\mathcal{G}$. □

Another decidable fragment of safety connectivity games arises from restricting Constructor in such a way that she cannot move any strong node. Since Constructor is not able to restore any deleted node, we can ignore the deleted nodes in each network. Hence, we only have to explore a finite state space which is at most exponential in the size of the given game.

Theorem 3.6. *In the case that Constructor does not have any movement rule, solving safety connectivity games is in* ExpTime.

Proof. Consider a safety connectivity game $\mathcal{G}$ without movement rules. We transform $\mathcal{G}$ into an *infinite two-player game* (see [37, 38]) on a game graph G' such that each vertex of G' corresponds to a network of $\mathcal{G}$ and an indication of which player acts next. The edges of G' are directed; they lead from networks where Constructor moves to networks where Destructor acts and vice versa according to the possible movements in $\mathcal{G}$.

Due to Remark 3.2 we can assume w.l.o.g. that Destructor plays according to a strict strategy. We assume that the initial network consist of $|V|$ active nodes and $|S|$ strong nodes. Then, the level decreases at most $|V| - |S|$ times before Destructor wins or all nodes in the network are strong.

Constructor is not able to restore any deactivated node; thus, we reduce the state space by ignoring the deactivated nodes in each network. Assuming that Destructor never skips after a node creation and ignoring deactivated nodes, the number of different networks of the same level is at most exponential in $\mathcal{G}$; the number of different levels that we have to consider is linear in $\mathcal{G}$. So, the size of the game graph G' is at most exponential in $\mathcal{G}$; hence, we can compute G' in exponential time. Solving the safety connectivity game $\mathcal{G}$ is equivalent to solving the safety game on the unfolded game graph G', which is feasible in linear time with respect to the size of the given game graph G' (see [36, 37, 38]). □

3.3. Non-Expanding and Unlabeled Non-Expanding Games

We have already showed in Theorem 3.5 that we can solve safety connectivity games in PSPACE if weak creation rules are forbidden. In the following we show that this lower bound cannot be improved: solving safety connectivity games is PSPACE-hard even in the more restricted unlabeled non-expanding case.

In order to show the PSPACE-hardness result, we use a polynomial-time reduction from solving *sabotage games*, which we mentioned in the introduction. Löding and Rohde showed that the problem of solving sabotage games is complete for the class PSPACE [6, 7].

Here, we define a *sabotage game* as a tuple

$$\mathcal{G}_s = (G_s, v_{\text{in}}, F)$$

where $G_s = (V, E)$ is a game graph, which changes over time, v_{in} is the initial vertex, and $F \subseteq V$ a set of final vertices. The two players, called *Runner* and *Blocker*, play as follows. Runner starts a play in v_{in} and moves to a vertex v' with $(v_{\text{in}}, v') \in E$. Then, Blocker erases an edge $e \in E$ resulting in a graph $G'_s = (V, E \setminus \{e\})$. Runner continues the play on G'_s, and so on in alternation. Runner wins iff she has a strategy to reach a vertex in F. The *problem of solving sabotage games* is the question of whether Runner wins for a given $\mathcal{G}_s$.

Löding and Rohde introduced sabotage games on multigraphs (i.e., graphs with possibly multiple edges between pairs of vertices), but showed that graphs with single edges suffice (see Lemma 1 in [6]). Now we can show our PSPACE-hardness result by simulating sabotage games.

Theorem 3.7. *In the unlabeled non-expanding case, solving safety connectivity games is* PSPACE*-hard.*

Proof. We use a polynomial-time reduction from the problem of solving sabotage games, which is PSPACE-hard [6], to the problem of solving unlabeled non-expanding safety games (i.e., safety games where all nodes are labeled with $\sqcup$ and the only rule for Constructor is $\langle \sqcup \xrightarrow{\text{move}} \sqcup \rangle$). For every sabotage game $\mathcal{G}_s = (G_s, v_{\text{in}}, F)$ on a game graph $G_s = (V_s, E_s)$, we construct an unlabeled non-expanding game $\mathcal{G}$ such that Constructor can preserve the connectivity of the network in $\mathcal{G}$ iff Runner wins the sabotage game $\mathcal{G}_s$.

The idea is to simulate each move of the Runner by the moving a strong node. Intuitively, Constructor should be able to move a strong node to a "final vertex" in the connectivity game exactly if Runner reaches a corresponding final vertex in the sabotage game. Only in this case Constructor can prevent Destructor from destroying the connectivity. We will ensure this by connecting the final vertices to a complete graph, which we denote by $K_{|V_s|}$, consisting of $|V_s|$ nodes. Each final vertex has an edge to every node of $K_{|V_s|}$, and every node of $K_{|V_s|}$ is also connected with an additional *target node*, denoted by v_{target}. Destructor will be able to isolate the target node by deleting all nodes of the subgraph $K_{|V_s|}$ if Constructor cannot move a strong node to $K_{|V_s|}$. For the remaining network Constructor can always guarantee the connectivity.

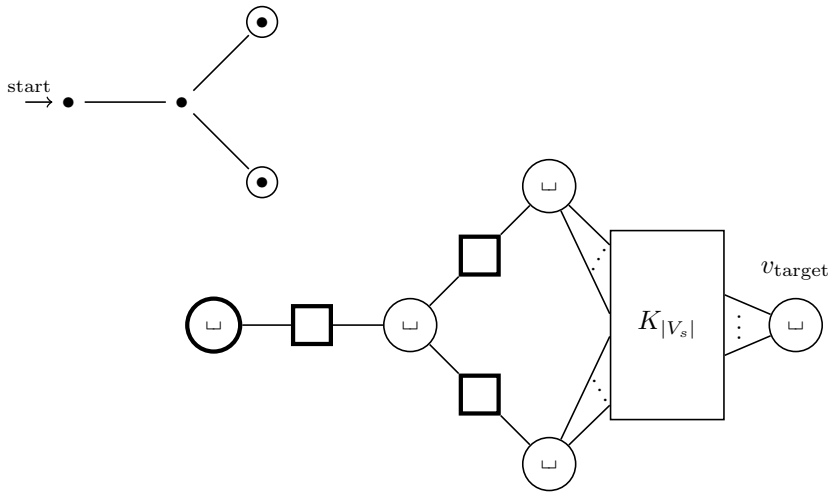

Figure 7: A game graph G_s of a sabotage game and its corresponding initial network G of a safety game. Bold squares are the replacement gates for the edges of G_s.

In order to realize the construction, we replace the edges of G_s by so-called *gates*, which we depict by a bold square. Figure 7 shows the initial graph G_s of an example sabotage game and its equivalent initial network G of our safety game. The replacement gate for an edge between two nodes u and v is depicted in Figure 8. All gates in the network share the same vertices z_1 and z_2, and each node of the complete subgraph $K_{|V_s|}$ is connected to z_1 as well. Constructor simulates a move of Runner from u to v by moving the strong node at w to v without giving Destructor the opportunity to isolate one of the nodes x_i. For this Constructor needs a strong node at u that she can move to w in case Destructor deletes a y_i-node.

Formally, we define the initial network as $G = (V, E, A, S, (P_\sqcup))$ with $V \coloneqq V_s \cup E_s \cup \{w, x_1, x_2, y_1, y_2\} \times E_s \cup \{z_1, z_2\} \cup V(K_{|V_s|}) \cup \{v_{\text{target}}\}$, $S \coloneqq \{v_{\text{in}}, z_1\} \cup \{w\} \times E_s$, and $P_\sqcup \coloneqq V$, where $V(K_{|V_s|})$ is the set of vertices of the complete subgraph $K_{|V_s|}$. The set E of edges arises from E_s as in the figures; additionally, each node in $K_{|V_s|}$ is connected to the vertex v_{target}, to every vertex in F, and to the z_1-vertex of each gate. It is easy to see that the constructed network G is only polynomial in the size of the sabotage game $\mathcal{G}_s$.

In this proof we assume Constructor starts the game (alike Runner starts the sabotage game). One can easily construct an equivalent game where Destructor starts: connect the strong node of G that corresponds to the initial vertex in $\mathcal{G}_s$ with one of the final vertices via an additional gate; then, Destructor has to delete a y_i-node of this gate in the first turn.

In order to show the correctness of the construction, we assume that Runner has a winning strategy in the sabotage game $\mathcal{G}_s$. We know that then Runner can also win $\mathcal{G}_s$ without visiting any vertex twice [8]. Therefore, we assume that Runner wins within $|V_s| - 1$ moves. Whenever Runner moves in $\mathcal{G}_s$ from

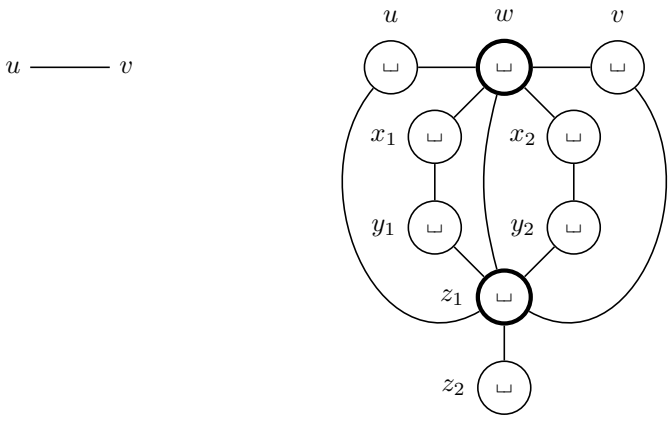

Figure 8: An edge between two nodes u and v in the sabotage game and its replacement gate.

a node u to a node v, Constructor shifts the strongness from w to v in the replacement gate for the edge (u, v). If Destructor deletes one of the nodes y_1, y_2, or w in this gate later on, Constructor reacts by securing this gate by moving the strong node from u to w (otherwise this move is not necessary). Since Runner reaches a final vertex in $\mathcal{G}_s$ within $|V_s| - 1$ moves, Constructor is able to move a strongness to the associated final node in the connectivity game within $|V_s| - 1$ moves, i.e., the network is still connected after the following move of Destructor. Then, Constructor can move this strong node to a node of $K_{|V_s|}$. From this point onwards Constructor can keep the network connected by securing the gates by moving the strong nodes from u-nodes to w-nodes as described before. Hence, Constructor wins $\mathcal{G}$.

Conversely, assume that Blocker has a winning strategy in $\mathcal{G}_s$. For each deletion of an edge (u, v) in $\mathcal{G}_s$, Destructor deletes a y_i-node of the associated gate in the connectivity game. In the case that Constructor shifts the strongness from a w-node to a v-node without having the required strongness at the associated u-node, Destructor reacts by deleting a y_i-node of this gate (and thereafter the w-node if Constructor does not shift the strongness back to w). If Constructor shifts a strongness from a w-node to a x_i-node, Destructor tries to isolate the other x_i-node. In the case that Constructor moves the strong node at z_1, Destructor wins immediately by deleting z_1. Since Runner loses the sabotage game, Constructor cannot move a strong node to the subgraph $K_{|V_s|}$ without allowing Destructor to disconnect he network. After blocking the replacement gates according to Blocker's winning strategy in $\mathcal{G}_s$, Destructor can delete all vertices of $K_{|V_s|}$. In this case v_{target} becomes isolated. Hence, Destructor wins $\mathcal{G}$. □

So, solving safety connectivity games is PSPACE-complete also in the cases where we only consider non-expanding and unlabeled non-expanding games.

4. Solvability of Reachability Connectivity Games

This section deals with the problem of solving reachability connectivity games. First we analyze the general case, for which we obtain undecidability even with a restricted rule set (Section 4.1). Then we discuss decidable subcases. For solving these reachability games we obtain an ExpTime upper bound and a PSpace lower bound (Section 4.2). Finally, we improve these results for unlabeled non-expanding games; in this case we prove a PSpace upper bound and an NP lower bound (Section 4.3).

4.1. The General Case

Solving reachability connectivity games is also undecidable in general. In contrast to the undecidability result for the safety connectivity games, we obtain the undecidability for reachability games also with restricted rule sets. In the reachability game we simulate a Turing machine solely by Constructor, who may connect a network if a stop state is reached (whereas in the safety game Constructor has to simulate transitions in order to compensate Destructor's node deletions). As a consequence Constructor can simulate a Turing machine also in a game where the network consists of strong nodes only, and Constructor modifies the cell nodes only with strong creation and relabeling rules. Alternatively, we can also use the idea of the proof of Theorem 3.1 to relabel adjacent cell nodes with weak creation rules and guarantee with movement rules that only these two adjacent cell nodes are strong. In both cases node deletions do not affect the reduction; so, solving reachability connectivity games remains undecidable even for the solitaire game version where Destructor always skips.

Theorem 4.1. *Solving reachability connectivity games is undecidable even if Constructor can only apply strong creation and relabeling rules or she can only apply weak creation and movement rules. In both cases the problem remains undecidable in the solitaire game version where Destructor never moves.*

Proof. We first describe the proof for the case that Constructor can only apply strong creation and relabeling rules; later we describe the modifications that are necessary for the case that Constructor is restricted to weak creation and movement rules.

We use a reduction from the halting problem for Turing machines. Again, we present Turing machines in the format

$$M = (Q, \Gamma, \delta, q_0, q_{\text{stop}})$$

with a state set Q, a tape alphabet Γ (including a blank symbol $\sqcup$), a transition function $\delta\colon Q \setminus \{q_{\text{stop}}\} \times \Gamma \to Q \times \Gamma \times \{L, R\}$, an initial state q_0, and a stop state q_{stop}.

For each such Turing machine M we construct a reachability game $\mathcal{G} = (G, R)$ such that M halts when started on the empty tape iff Constructor can reach a connected network from the initial network G by applying the rules of R.

The idea of the construction is to represent a Turing machine configuration $a_1 \dots a_{i-1}(q, a_i)a_{i+1} \dots a_n$ by a sequence of n nodes. We label the nodes for such a sequence with pairs from $\Gamma \times (Q \cup \{\mathfrak{l}, \mathfrak{r}\})$. The first component holds the entry of its corresponding cell of the tape. The second component is labeled with $q \in Q$ if M is in state q and the head is on the cell represented by this node; otherwise the label $\mathfrak{l}$ or $\mathfrak{r}$ denotes whether the represented cell is on the left-hand side or right-hand side of the head. Additionally, the label alphabet contains the symbols], $\top$, $\bot$, and $+$, where] is an end marker connected to the rightmost cell of the tape, $\top$ is the label of a disconnected node, $\bot$ is the label of an anchor node that is connected to all nodes except the $\top$-labeled node, and $+$ is only used as a dummy label for the node that Constructor creates to connect the nodes labeled with $\top$ and with $\bot$ in case M stops.

As initial network we take a four node graph. Two connected nodes labeled $(\sqcup, q_0)$ and] represents M in its initial state q_0 on the empty tape. The third node is labeled with $\top$ and is disconnected from any other node. The fourth node labeled $\bot$ is connected to any other node except the $\top$-labeled node. We define all of these nodes as strong.

It is easy to supply a set of relabeling rules which allow to change the network only in a way that the computation of M is simulated. Formally, for each tuple (q, a, p, b, X) with $\delta(q, a) = (p, b, X)$ and for each $y \in \Gamma$ we add the rule

$$\langle\, (y, \mathfrak{l}), (a, q) \longrightarrow (y, q), (b, \mathfrak{r}) \,\rangle$$

if $X = L$, and

$$\langle\, (a, q), (y, \mathfrak{r}) \longrightarrow (b, \mathfrak{l}), (y, q) \,\rangle$$

if $X = R$. For the case that more space on the tape is needed (beyond the current end marker), we introduce a strong creation rule that extends the network by relabeling the current end marker to a cell node (carrying the blank symbol) and creating a new end marker that is connected to the former end marker and also to the $\bot$-labeled anchor node:

$$\langle\,], \bot \xrightarrow{\text{s-create}(\,])} (\sqcup, \mathfrak{r}), \bot \,\rangle \ .$$

Finally, we allow a special creation rule that can be applied when the stop state q_{stop} is reached:

$$\langle\, q_{\text{stop}}, \top \xrightarrow{\text{s-create}(+)} q_{\text{stop}}, \top \,\rangle \ .$$

Constructor can apply this rule if M halts; only in this case a connected network is reached.

Since in our simulation we have defined all nodes as strong, Destructor is never able to delete any node. Hence, the undecidability results also hold for the solitaire version of the game where Destructor never moves.

In the following we provide a modified version of the construction for the case that Constructor can only apply weak creation and movement rules. We

define the initial network exactly as before, but now we supply a set of weak creation rules to simulate the computation of the Turing machine. Formally, for each tuple (q, a, p, b, X) with $\delta(q, a) = (p, b, X)$ and for each $y \in \Gamma$ we add the rule

$$\langle\, (y, \mathfrak{l}), (a, q) \xrightarrow{\text{create}(+)} (y, q), (b, \mathfrak{r})\,\rangle$$

if $X = L$, and

$$\langle\, (a, q), (y, \mathfrak{r}) \xrightarrow{\text{create}(+)} (b, \mathfrak{l}), (y, q)\,\rangle$$

if $X = R$. In order to allow Constructor only to simulate valid transitions, only the cell node labeled with the current state and one adjacent cell node are strong (which correspond in the initial network the cell node and the end marker node). This ensures that Constructor can apply an accordant creation rule only to these two cell nodes. To simulate a transition Constructor has to shift the two strong nodes to the desired positions. Formally, we add all movement rules of the form

$$\langle\, u \xrightarrow{\text{move}} v\,\rangle$$

where either $u \in \Gamma \times Q$, $v \in \Gamma \times \{\mathfrak{l}, \mathfrak{r}\} \cup \{\,]\,\}$, or $u \in \Gamma \times \{\mathfrak{l}, \mathfrak{r}\} \cup \{\,]\,\}$, $v \in \Gamma \times Q$. These rules guarantee that the two strong nodes are adjacent whenever one of the nodes carries the current state. Again, we have the rule

$$\langle\,], \bot \xrightarrow{\text{create}(]\,)} (\llcorner\!\lrcorner, \mathfrak{r}), \bot\,\rangle$$

to extend the tape and the rule

$$\langle\, q_{\text{stop}}, \top \xrightarrow{\text{create}(+)} q_{\text{stop}}, \top\,\rangle$$

to connect the network if the stop state is reached.

In this modified construction relabeling rules are absent. Also, Destructor cannot disconnect originally connected nodes by node deletion. Hence, the undecidability results also hold for the solitaire version of the game where Destructor never moves. □

4.2. Decidable Subcases

As shown by the previous theorem the undecidability of solving expanding reachability games rely on the availability of creation moves. If these are omitted, the state space is finite and hence the problem becomes trivially decidable.

Remark 4.2. In the non-expanding case, solving reachability connectivity games is in ExpTime.

Proof. We transform $\mathcal{G}$ into an *infinite two-player game* (see [37, 38]) on a game graph G' such that each vertex of G' corresponds to a network of $\mathcal{G}$ and an indication of which player acts next. The edges of G' lead from networks where Constructor moves to networks where Destructor acts and vice versa according to the possible movements in $\mathcal{G}$.

Since $\mathcal{G}$ is non-expanding, the size of G' is at most exponential in $\mathcal{G}$, and we can compute G' in exponential time. Solving the reachability connectivity game $\mathcal{G}$ is equivalent to solving the reachability game on the unfolded game graph G', which is feasible in linear time with respect to the size of the given game graph G' (see [36, 37, 38]). □

Complementary to this ExpTime upper bound, we provide a PSpace lower bound. We prove the lower bound by a reduction from the problem of solving sabotage games, which we briefly defined at the beginning of Section 3.3.

Theorem 4.3. *In the non-expanding case, solving reachability connectivity games is* PSpace*-hard.*

Proof. We use a polynomial-time reduction from the problem of solving sabotage games, which is PSpace-hard [6], to the problem of solving non-expanding reachability connectivity games. For every sabotage game $\mathcal{G}_s = (G_s, v_{\text{in}}, F)$ on a game graph $G_s = (V_s, E_s)$, we construct a non-expanding game $\mathcal{G} = (G, R)$ such that Constructor can reach a connected network in $\mathcal{G}$ iff Runner wins the sabotage game $\mathcal{G}_s$.

The idea is to allow Constructor to propagate a label through the graph according to Runner's movement in the sabotage game. For that purpose we use the node labels 'vertex', 'edge', 'run', 'final', 'reach', $\top$, and $\bot$. Each vertex of the sabotage game becomes 'vertex'-labeled strong node in the initial network of the connectivity game except the initial vertex, which is labeled 'run', and the final vertices, which are labeled 'final'. We represent each edge of the sabotage game by a weak intermediate nodes labeled 'edge'. We simulate Runner's movements by relabeling rules of the form $\langle\, \text{run}, * \longrightarrow *, \text{run} \,\rangle$. Each of Blocker's edge removal corresponds the deletion of an intermediate 'edge'-labeled node. If the 'run' label reaches the 'final' label, it is relabeled to 'reach'. The network contains an isolated $\top$-labeled strong node, which is only connected to the remaining network via a deactivated $\bot$-labeled node. More precisely, the deactivated $\bot$-labeled node is adjacent to all strong nodes. In the case of a successful simulation, where the 'run' label is relabeled to a 'reach' label, Constructor can reach a connected network by moving the 'reach'-labeled strong node to the deactivated $\bot$-labeled node.

The network G is defined as described. Figure 9 shows the initial game graph G_s of an example sabotage game and the equivalent initial network G of our game. The rule set R consists of the following three rules. To simulate Runner's moves the rule

$$\langle\, \text{run}, \text{edge} \longrightarrow \text{vertex}, \text{run} \,;\, \text{run}, \text{vertex} \longrightarrow \text{edge}, \text{run} \,\rangle$$

allows Constructor to propagate the 'run' label from one 'vertex'-labeled node to another by passing one non-deleted 'edge'-labeled node. We add a second rule that additionally relabels a node from 'final' to 'reach' if it is reached by the 'run' label:

$$\langle\, \text{run}, \text{edge} \longrightarrow \text{vertex}, \text{run} \,;\, \text{run}, \text{final} \longrightarrow \text{edge}, \text{reach} \,\rangle \ .$$

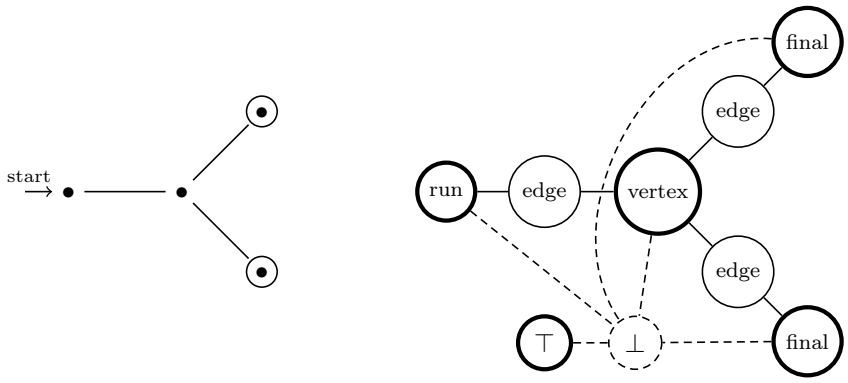

Figure 9: A game graph G_s of a sabotage game and its corresponding initial network G of a reachability game.

The third rule

$$\langle\, \text{reach} \xrightarrow{\text{move}} \bot \,\rangle$$

allows Constructor to connect the network immediately if a node carries the 'reach' label.

In this proof we assume Constructor starts the game (alike Runner starts the sabotage game). One can easily construct an equivalent game where Destructor starts: connect the 'run'-labeled node to one 'final'-labeled node via an additional 'edge'-labeled node; then, Destructor has to delete this extra 'edge'-labeled node in the first turn.

Since applying the third rule is the only way for Constructor to connect $\top$-labeled node with the remaining network, Constructor can reach a connected network iff she can reach a network containing a 'reach' label. Hence, Constructor is able to reach a connected network in $\mathcal{G}$ iff Runner wins the sabotage game $\mathcal{G}_s$. □

We have seen that solving connectivity reachability games becomes decidable if we forbid any creation of nodes. However, we can also identify a decidable fragment that still allows the creation of weak nodes. More precisely, reachability games are also decidable if we only allow weak creation and relabeling rules. These games lack the dynamics that arises from modifying the set of strong nodes. In particular, a new created weak node can never be isolated from its strong adjacent nodes. As a consequence we can assume that Destructor never skips after a creation step of Constructor (instead of skipping, he can deactivate the weak node that has just been created). This allows us to adapt the proof of Theorem 3.6, where we reduce the state space by ignoring the deactivated nodes.

Theorem 4.4. *In the case that Constructor does neither have strong creation rules nor movement rules, solving reachability connectivity games is in* ExpTime.

Proof. Consider a reachability game $\mathcal{G}$ without strong creation and movement rules. Again, we transform the game $\mathcal{G}$ into an *infinite two-player game* on a game graph G' such that each vertex of G' corresponds to a network of $\mathcal{G}$ and an indication of which player acts next (see Theorem 3.6 and Remark 4.2).

Since the set of strong nodes is fixed throughout every play, every node that Constructor creates with a weak creation rule is only adjacent to strong nodes. Destructor is not able to isolate this node from its adjacent nodes. Hence, if a network G where such a created node u is active is disconnected and Destructor has a winning strategy from G, Destructor also wins from the network that arises from G by deactivating u. Thus, we can assume w.l.o.g. that Destructor never skips after Constructor creates a node (in any case it is better for Destructor to delete the created node than to skip).

Constructor is not able to restore any deactivated node; thus, we reduce the state space by ignoring the deactivated nodes in each network. Assuming that Destructor never skips after a node creation and ignoring deactivated nodes, the number of different networks is at most exponential in $\mathcal{G}$. So, the size of G' is at most exponential in $\mathcal{G}$; hence, we can compute G' in exponential time. Solving the reachability connectivity game $\mathcal{G}$ is equivalent to solving the reachability game on the unfolded game graph G', which is feasible in linear time with respect to the size of the given game graph (see [36, 37, 38]). □

Also for this fragment we obtain a PSPACE lower bound.

Theorem 4.5. *In the case that Constructor does neither have strong creation rules nor movement rules, solving reachability connectivity games is* PSPACE*-hard.*

Proof. We reuse the reduction presented in the proof of Theorem 4.3. In this construction we only have to replace the movement rule $\langle\, \text{reach} \xrightarrow{\text{move}} \bot \,\rangle$, which lets Constructor connect the network in the case that the Runner reaches a final vertex in the sabotage game. To achieve this we introduce instead the weak creation rule

$$\langle\, \text{reach}, \top \xrightarrow{\text{create}(\bot)} \text{reach}, \top \,\rangle \ ,$$

which lets Constructor connected the isolated $\top$-labeled node with the remaining network if a vertex obtains the 'reach' label. □

4.3. Unlabeled Non-Expanding Games

For unlabeled non-expanding games, in which Constructor can only move strong nodes, we give an NP lower bound and a PSPACE upper bound.

Theorem 4.6. *In the unlabeled non-expanding case, solving reachability connectivity games is* NP*-hard.*

Proof. We use a polynomial-time reduction from the *vertex cover* problem, which is well-known to be NP-hard. We state the vertex cover problem in the following form: given a graph $G_{\text{VC}} = (V', E')$ and an integer k, is there a

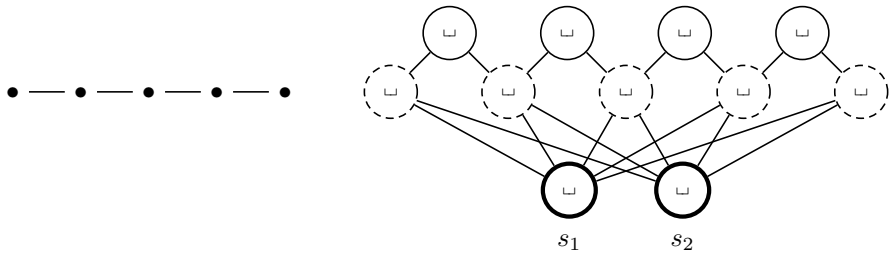

Figure 10: A graph G_{VC} and the corresponding initial network G of a reachability game for testing G_{VC} for a vertex cover of size two.

vertex cover of G_{VC} of at most k vertices (i.e., is there a set $C \subseteq V'$ of at most k vertices such that each edge of G_{VC} is incident to at least one vertex in C)?

We construct an unlabeled non-expanding game $\mathcal{G}$, i.e., the only node label is $\sqcup$ and the only rule for Constructor is $\langle \sqcup \xrightarrow{\text{move}} \sqcup \rangle$. The idea is to modify G_{VC} (by adding an intermediate node for each edge) in order to use the graph as the initial network of a connectivity game. Then, Constructor will only be able to reach a connected network by moving k strong nodes to vertices that form a vertex cover of G_{VC}.

More precisely, the initial network G results from G_{VC} by adding a weak node for each edge. We keep the original nodes of G_{VC} as deactivated nodes. Additionally, each of these deactivated nodes is connected with k strong nodes $s_1, \ldots, s_k$. For example the graph G_{VC} in Figure 10 has a vertex cover of size two. So, two strong nodes are sufficient for Constructor to preserve the connectivity in the corresponding network G.

If for G_{VC} a vertex cover with at most k vertices exists, player Constructor wins by moving strong nodes to the vertex cover. In this case each of the nodes that corresponds to an edge in G_{VC} is connected to a strong node. Note that all strong nodes are still connected via the vertex s_i from which Constructor shifts the last strongness to the vertex cover; Destructor can only delete s_i in the next turn.

For the converse note that it is best for Constructor to move strong nodes to nodes that correspond to vertices in G_{VC}; if Constructor shifts such a strongness again to any other node, Destructor immediately deletes the vertex where this strongness was shifted from. So, if Constructor wins the reachability game $\mathcal{G}$, the vertices to that Constructor moves the strong nodes corresponds to a vertex cover in G_{VC}. □

Now, we establish a PSPACE upper bound for the unlabeled non-expanding case. The basic observation is the following. If Constructor moves some strong node a certain number of times, she moves a strong node in a loop that cannot be necessary for a winning strategy. For this purpose, we first note an upper bound on the number of moves of a strong node; we know that after $k \cdot |V|$ moves Constructor has shifted this strongness in some loop at least k times starting

from a certain vertex.

Remark 4.7. If Constructor shifts some strongness $k \cdot |V|$ times, there is a vertex $v \in V$ that this strongness visits $k+1$ times, i.e., the strongness is shifted through k loops that start and end at v.

We show that Constructor does not need to shift a strong node through more than $2 \cdot |V| - 2$ loops starting from the same vertex. Then, we can infer from the previous remark that it is sufficient for Constructor to shift each strongness at most $2 \cdot |V|^2 - 1$ times.

Lemma 4.8. *Consider an unlabeled non-expanding reachability game $\mathcal{G}$. If Constructor wins $\mathcal{G}$, she also wins $\mathcal{G}$ with a strategy where she shifts each strongness at most $2 \cdot |V|^2 - 1$ times.*

Proof. Let us assume that Constructor has a winning strategy σ but that, towards a contradiction, Destructor has a strategy τ such that Constructor has to shift some strongness at least $2 \cdot |V|^2$ times before she wins. Consider a play π where Destructor and Constructor play according to σ and τ, respectively. Then, the previous remark states that there is a vertex $v \in V$ from which Constructor moves some strong node through at least $2 \cdot |V|$ loops before she wins the play π.

In a reachability game where only movement rules are allowed, Destructor cannot restrict Constructor's possibilities to move. Hence, there are only two possible reasons for Constructor to move the mentioned strong node in a loop that starts and ends at v.

1. Some node $u \in V \setminus \{v\}$ is restored by moving the strong node in that loop. However, in this case we can assume that Constructor does not restore u again while shifting the strongness in a loop that starts and ends at v. Otherwise Constructor can omit each former loop in which she moves the strong node only for this reason; Constructor still wins with this modified strategy.
2. Destructor deletes some node $u \in V \setminus \{v\}$ while Constructor moves the strong node in that loop. (Here, we consider this as an achievement for Constructor, e.g., it may be that Destructor loses during this loop if he does not delete u.) Also in this case we can assume that Constructor does not shift again this strongness in a loop that starts and ends at v. If u is still deactivated, Constructor can clearly omit shifting this strongness in this loop only for the reason of letting Destructor delete u; if Constructor has restored u (before she wins), she can omit each former loop in which she shifts this strongness only for this reason. In either case Constructor wins with the modified strategy.

Since in each case $u \in V \setminus \{v\}$, we can assume that each of these cases occurs at most $|V| - 1$ times if Constructor plays optimal. Hence, we can optimize Constructor's winning strategy τ to a winning strategy τ' with which she shifts each strongness through at most $2 \cdot |V| - 2$ loops that start and end at the same vertex. (The optimization can be done iteratively, similarly as described at the

end of Lemma 3.3 for Destructor's strategy.) The obtained winning strategy τ' lead to a contradiction as we have shown already that in the before mentioned play π there must be a vertex $v \in V$ from which Constructor moves some strong node through at least $2 \cdot |V|$ loops before she wins. □

We lift the upper bound for the number of moves of each strong node (in reachability games where Constructor wins) to the overall number of moves that Constructor needs to win.

Lemma 4.9. *Consider an unlabeled non-expanding reachability game $\mathcal{G}$ where the network consists of $|V|$ vertices, $|S|$ of which are strong. If Constructor wins $\mathcal{G}$, she also has a strategy to win $\mathcal{G}$ with at most $2 \cdot |S| \cdot |V|^2 - 1$ moves.*

Proof. For connectivity games with a reachability objective we can assume that Constructor never skips: if Constructor skips, Destructor can skip as well leading the play to the same network (which is still disconnected). Since Constructor never skips, there exists a strongness that she shifts at least k times within $|S| \cdot k$ moves. By Lemma 4.8 Constructor wins with $2 \cdot |S| \cdot |V|^2 - 1$ moves if she has a winning strategy. □

To show the decidability in PSPACE it suffices to build up the game tree and truncate it after $2 \cdot |S| \cdot |V|^2 - 1$ moves of Constructor (analogously to Theorem 3.5).

Theorem 4.10. *In the unlabeled non-expanding case, solving reachability connectivity games is decidable in* PSPACE.

Proof. The proof is analogous to the proof of Theorem 3.5; we use the bound from Lemma 4.9 to truncate the game tree. □

5. Conclusion and Open Problems

In this work we introduced dynamic network connectivity games and studied the complexity of solving these games in both the reachability and the safety version. We showed that both problems are undecidable in general. Nevertheless, we pointed out decidable fragments by restricting the permitted rule types. For these fragments, we encountered fundamental differences in the decidability and the computational complexity of solving reachability and safety connectivity games. In particular, we investigated the complexity for non-expanding connectivity games, where node creation is forbidden. In this case we studied the fragment of unlabeled non-expanding games, where nodes are labeled uniformly. An overview of the results was given in Table 1 on page 5.

The careful reader may have noticed that we have not discussed every possible restriction of the rule set. Some of these missing cases are easy to analyze or are described already in a more general way with another restriction of the rule types (e.g., the games where only weak creation rules are allowed or only relabeling rules are allowed). Other cases seem to be challenging.

Problem 1. Are reachability connectivity games algorithmically solvable under the restrictions that

- only strong creation rules are allowed,
- only strong creation and weak creation rules are allowed, or
- only strong creation and movement rules are allowed?

Are safety connectivity games algorithmically solvable under the restriction that only weak creation and movement rules are allowed?

Our results leave a gap between the upper and the lower bound for the complexity of solving non-expanding reachability games. We conjecture that these are easier to solve in the unlabeled non-expanding case than in the more general non-expanding case. Note, for instance, that one can use relabeling rules to define a binary counter on some nodes; then, one may imagine a game in which Constructor has to perform an exponential number of relabeling steps before she can establish a connected network. This indicates that the number of turns that a player needs to win cannot be bounded by some polynomial, and hence the winner cannot be determined in PSpace. A proof, however, is still missing. Also for some other cases listed in Table 1 tight complexity bounds are missing.

Problem 2. Are reachability connectivity games in the non-expanding case harder to solve than in the unlabeled non-expanding case? Is it possible to provide tight complexity bounds for all cases listed in Table 1?

Some of our results depend on the balance between node deletion and restoration; if Constructor restores a node, Destructor can delete another one immediately. If one allows rules for multiple movements and relabelings in Constructor's turns, the complexity of solving connectivity games increases. In the non-expanding case, one can show that solving both reachability and safety games becomes ExpTime-complete [25]. The hardness result can be obtained via a reduction from the halting problem of polynomial space-bounded alternating Turing machines. It is less clear what happens in the expanding case where the solvability depends on this balance.

Problem 3. If one adds rules to allow multiple movements and relabelings for Constructor, to what extent do the results in this article still hold?

We only considered dynamic network connectivity games with reachability and safety specifications. In practice one may consider a more involved *recurrence (Büchi) condition*, where Constructor has to reach a connected network again and again, or a *persistence (co-Büchi) condition*, where Constructor has to guarantee that the network stays connected from some point onwards. We expect at least that the negative results in this article (undecidability and hardness) for reachability connectivity games also hold for connectivity games with a recurrence condition and that the negative results for safety connectivity games

also hold for connectivity games with a persistence condition. It is, however, not clear whether we can adapt the positive results (i.e., the upper bounds for solving some games) to connectivity games with recurrence and persistence conditions.

Problem 4. To what extent can the results in this article be generalized from reachability and safety connectivity games to Büchi and co-Büchi connectivity games, respectively?

Instead of games with reachability, safety, Büchi, and co-Büchi winning conditions, one may consider properties in *linear temporal logic* (LTL), in which one can express all of the before mentioned conditions. A generalization in the context of connectivity games are LTL specifications over a single predicate that is true in move i iff the current network is connected in move i. For non-expanding games with such an LTL winning condition we know an ExpTime lower and a 2ExpTime upper bound [1]. The upper bound is obtained by reducing these games to LTL games on graphs, whose solvability has been shown to be complete for 2ExpTime [40] (also see [41, 42]). The lower bound is shown by a reduction from polynomial space-bounded alternating Turing machines; it uses the power of LTL to force Destructor to delete nodes in certain turns and to prevent him from deleting nodes in other turns.

Finally, we mention possible refinements of the model and the problem statement. In the view of verifying fault-tolerant systems it is rarely realistic to assume an omniscient adversary who deletes nodes; faults are better modeled as random events. This scenario has been studied in the framework of sabotage games [9]. One can study the corresponding case for connectivity games, where each action of Destructor is replaced by a random vertex deletion [1].

Indeed, our game model needs also to be refined for studying routing problems. To this end one may pursue two approaches. The first approach is to extend the winning conditions only. We can take the model of dynamic network connectivity games (as defined in this article) and refine the winning conditions with an additional constraint that requires that certain labels are propagated with relabeling rules from their source nodes to their destination nodes. The second approach is to extend the model with packets, which has to be delivered to their destinations, as in [2] (or [25], Chapter 2). In fact, the second approach results in a complex and powerful model, which can handle an unbounded number of packets, whereas the first approach may be probably allow better algorithmic solutions.

Also, the problem of solving games in the form of a decision problem (i.e., the question of whether a given specification is satisfied or not) has to be refined. From a practical point of view it is more useful to formulate the problem as an optimization problem, where we ask, for instance, how many strong nodes are necessary to guarantee the connectivity of the network. For this optimization problem one can use simple heuristics as studied in [1]. These yield small (although not optimal) solutions with efficient winning strategies on various classes of networks.

References

[1] S. Grüner, Game Theoretic Analysis of Dynamic Networks, Diploma thesis, RWTH Aachen, 2011.

[2] J. Gross, F. G. Radmacher, W. Thomas, A game-theoretic approach to routing under adversarial conditions, in: Proceedings of IFIP TCS, volume 323 of *IFIP Advances in Information and Communication Technology*, Springer, 2010, pp. 355–370.

[3] J. H. Reif, The complexity of two-player games of incomplete information, Journal of Computer and System Sciences 29 (1984) 274–301.

[4] S. Azhar, G. L. Peterson, J. H. Reif, Lower bounds for multiplayer non-cooperative games of incomplete information, Journal of Computers and Mathematics with Applications 41 (2001) 957–992.

[5] J. van Benthem, An essay on sabotage and obstruction, in: Mechanizing Mathematical Reasoning: Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday, volume 2605 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 268–276.

[6] C. Löding, P. Rohde, Solving the Sabotage Game is PSPACE-hard, Technical Report AIB-05-2003, RWTH Aachen, 2003.

[7] C. Löding, P. Rohde, Solving the sabotage game is PSPACE-hard, in: Proceedings of MFCS, volume 2747 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 531–540.

[8] P. Rohde, On Games and Logics over Dynamically Changing Structures, Ph.D. thesis, RWTH Aachen, 2005.

[9] D. Klein, F. G. Radmacher, W. Thomas, Moving in a network under random failures: A complexity analysis, Science of Computer Programming 77 (2012) 940–954.

[10] N. Gierasimczuk, L. Kurzen, F. R. Velázquez-Quesada, Learning and teaching as a game: A sabotage approach, in: Proceedings of LORI, volume 5834 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 119–132.

[11] L. Kurzen, Complexity in Interaction, Ph.D. thesis, Institute for Logic, Language and Computation, Amsterdam, 2011.

[12] E. Altman, T. Boulogne, R. El-Azouzi, T. Jiménez, L. Wynter, A survey on networking games in telecommunications, Computers & Operations Research 33 (2006) 286–311.

[13] A. Fiat, G. J. Woeginger, Online Algorithms: The State of the Art, volume 1442 of *Lecture Notes in Computer Science*, Springer, 1998.

[14] A. Borodin, R. El-Yaniv, Online Computation and Competitive Analysis, Cambridge University Press, 1998.

[15] S. Albers, Online algorithms: a survey, Mathematical Programming 97 (2003) 3–26.

[16] R. Rajaraman, Topology control and routing in ad hoc networks: a survey, SIGACT News 33 (2002) 60–73.

[17] C. Scheideler, Models and techniques for communication in dynamic networks, in: Proceedings of STACS, volume 2285 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 27–49.

[18] B. Awerbuch, P. Berenbrink, A. Brinkmann, C. Scheideler, Simple routing strategies for adversarial systems, in: Proceedings of FOCS, IEEE Computer Society, 2001, pp. 158–167.

[19] B. Awerbuch, A. Brinkmann, C. Scheideler, Anycasting in adversarial systems: Routing and admission control, in: Proceedings of ICALP, volume 2719 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 1153–1168.

[20] C. Demetrescu, D. Eppstein, Z. Galil, G. F. Italiano, Dynamic graph algorithms, in: M. J. Atallah, M. Blanton (Eds.), Algorithms and Theory of Computation Handbook, Second Edition, Volume 1: General Concepts and Techniques, CRC Press, 2010, pp. 9–1–9–28.

[21] J. Feigenbaum, S. Kannan, Dynamic graph algorithms, in: K. H. Rosen (Ed.), Handbook of Discrete and Combinatorial Mathematics, CRC Press, 2000, pp. 1142–1151.

[22] J. Holm, K. de Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity, Journal of the ACM 48 (2001) 723–760.

[23] L. Roditty, U. Zwick, A fully dynamic reachability algorithm for directed graphs with an almost linear update time, in: Proceedings of STOC, ACM, 2004, pp. 184–191.

[24] V. Weber, T. Schwentick, Dynamic complexity theory revisited, Theory of Computing Systems 40 (2007) 355–377.

[25] F. G. Radmacher, Games on Dynamic Networks: Routing and Connectivity, Ph.D. thesis, RWTH Aachen, 2012.

[26] R. Pucella, V. Weissman, Reasoning about dynamic policies, in: Proceedings of FoSSaCS, volume 2987 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 453–467.

[27] S. Demri, A reduction from DLP to PDL, Journal of Logic and Computation 15 (2005) 767–785.

[28] S. Göller, M. Lohrey, Infinite state model-checking of propositional dynamic logics, in: Proceedings of CSL, volume 4207 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 349–364.

[29] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific, 1997.

[30] A. Corradini, GETGRATS: A summary of scientific results (with annotated bibliography), in: Proceedings of GETGRATS Closing Workshop, volume 51 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2001, pp. 1–61.

[31] R. Heckel, Graph transformation in a nutshell, in: Proceedings of FoVMT, volume 148(1) of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2006, pp. 187–198.

[32] F. Gadducci, R. Heckel, M. Koch, A fully abstract model for graph-interpreted temporal logic, in: Proceedings of TAGT, volume 1764 of *Lecture Notes in Computer Science*, Springer, 1998, pp. 310–322.

[33] O. Kupferman, M. Y. Vardi, P. Wolper, Module checking, Information and Computation 164 (2001) 322–344.

[34] F. G. Radmacher, W. Thomas, A game theoretic approach to the analysis of dynamic networks, in: Proceedings of VerAS, volume 200(2) of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2008, pp. 21–37.

[35] S. Grüner, F. G. Radmacher, W. Thomas, Connectivity games over dynamic networks, in: G. D'Agostino, S. L. Torre (Eds.), Proceedings of GandALF, volume 54 of *Electronic Proceedings in Theoretical Computer Science*, 2011, pp. 131–145.

[36] W. Thomas, On the synthesis of strategies in infinite games, in: Proceedings of STACS, volume 900 of *Lecture Notes in Computer Science*, Springer, 1995, pp. 1–13.

[37] W. Thomas, Solution of church's problem: A tutorial, in: New Perspectives on Games and Interaction, volume 4 of *Texts in Logic and Games*, Amsterdam University Press, 2008, pp. 211–236.

[38] E. Grädel, W. Thomas, T. Wilke, Automata, Logics, and Infinite Games: A Guide to Current Research, volume 2500 of *Lecture Notes in Computer Science*, Springer, 2002.

[39] C. H. Papadimitriou, Computational Complexity, Addison Wesley, 1994.

[40] A. Pnueli, R. Rosner, On the synthesis of an asynchronous reactive module, in: Proceedings of ICALP, volume 372 of *Lecture Notes in Computer Science*, Springer, 1989, pp. 652–671.

[41] R. Rosner, Modular Synthesis of Reactive Systems, Ph.D. thesis, The Weizmann Institute of Science, 1991.

[42] R. Alur, S. La Torre, P. Madhusudan, Playing games with boxes and diamonds, in: Proceedings of CONCUR, volume 2761 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 127–141.